LASSO - an Observatorium for the Dynamic Selection, Analysis and Comparison of Software

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

M.Sc. Wirtschaftsinformatik Marcus Kessel
aus Weinheim

Mannheim, 2022

Dekan: Dr. Bernd Lübcke, Universität Mannheim
Referent: Prof. Dr. Colin Atkinson, Universität Mannheim
Korreferent: Prof. Dr. Uwe Aßmann, Technische Universität Dresden

Tag der mündlichen Prüfung: 16. Februar 2023

Abstract

Mining software repositories at the scale of "big code" (i.e., big data) is a challenging activity. As well as finding a suitable software corpus and making it programmatically accessible through an index or database, researchers and practitioners have to establish an efficient analysis infrastructure and precisely define the metrics and data extraction approaches to be applied. Moreover, for analysis results to be generalisable, these tasks have to be applied at a large enough scale to have statistical significance, and if they are to be repeatable, the artefacts need to be carefully maintained and curated over time. Today, however, a lot of this work is still performed by human beings on a case-by-case basis, with the level of effort involved often having a significant negative impact on the generalisability and repeatability of studies, and thus on their overall scientific value.

The general purpose, "code mining" repositories and infrastructures that have emerged in recent years represent a significant step forward because they automate many software mining tasks at an ultra-large scale and allow researchers and practitioners to focus on defining the questions they would like to explore at an abstract level. However, they are currently limited to static analysis and data extraction techniques, and thus cannot support (i.e., help automate) any studies which involve the execution of software systems. This includes experimental validations of techniques and tools that hypothesise about the behaviour (i.e., semantics) of software, or data analysis and extraction techniques that aim to measure dynamic properties of software.

In this thesis a platform called LASSO (Large-Scale Software Observatorium) is introduced that overcomes this limitation by automating the collection of dynamic (i.e., execution-based) information about software alongside static information. It features a single, ultra-large scale corpus of executable software systems created by amalgamating existing Open Source software repositories and a dedicated DSL for defining abstract selection and analysis pipelines. Its key innovations are integrated capabilities for searching for selecting software systems based on their exhibited behaviour and an "arena" that allows their responses to software tests to be compared in a purely data-driven way. We call the platform a "software observatorium" since it is a place where the behaviour of large numbers of software systems can be observed, analysed and compared.

Zusammenfassung

Das Mining von Software Repositorien in der Größenordnung von „Big Code“ (d.h. Big Data) stellt eine immense Herausforderung dar. Wissenschaftler und Praktiker müssen nicht nur passende Software Korpora erstellen, die über Datenbanken zugänglich sein müssen, sondern es ist auch ihre Aufgabe eine effiziente Analyseinfrastruktur zu entwickeln die es ermöglicht Metriken präzise zu definieren und Techniken zur Extraktion anzuwenden. Um Ergebnisse von Analysen verallgemeinern zu können, müssen Mining Aufgaben skalierbar sein um statistische Signifikanz zu erzielen. Aus den Korpora verwendete Artefakte sollten aktuell gehalten werden, sodass die Analyse Ergebnisse wiederholt werden können. Heutzutage werden diese Aufgaben meist immer noch von Menschen manuell durchgeführt was zur Folge hat, dass die Generalisierbarkeit von Ergebnissen und die Replikation von Studien eingeschränkt sind, wodurch diese an wissenschaftlichem Wert verlieren.

In den letzten Jahren wurden allgemeine Korpora und Infrastrukturen zum „Code Mining“ vorgestellt welche einen erheblichen Fortschritt darstellen da sie viele der verbundenen Mining Aufgaben in der gewünschten Größenordnung skalieren können. Wissenschaftler und Praktiker sind nun gezielt in der Lage Fragen über „Big Code“ zu stellen und diese auf einer abstrakten Ebene zu erforschen. Dennoch sind die Ansätze limitiert, da sie derzeit ausschließlich statische Analysen und Extraktionstechniken bereitstellen. Studien die die Ausführung von Software Systemen voraussetzen werden daher nicht unterstützt und können somit nicht automatisiert werden. Diese umfassen experimentelle Validierungen von Techniken und Werkzeugen welche das Verhalten (d.h. die Semantik) von Software nutzen, oder Datenanalysen und Extraktionstechniken welche zum Ziel haben dynamische Eigenschaften von Software zu messen.

In dieser Arbeit wird die Plattform LASSO („Large-Scale Software Observatorium“) vorgestellt, welche die genannten Einschränkungen durch die Sammlung von dynamischen, ausführungsbasierten Informationen beseitigt, sowie die Sammlung von statischen Informationen ermöglicht. Sie bietet einen einzelnen, skalierbaren Korpus von ausführbaren Software Systemen der Open Source Repositorien integriert, und eine dedizierte DSL um Selektions- und Analyse Prozesse definieren zu können. Die Schlüsselinnovationen bestehen aus der Fähigkeit Systeme anhand ihres tatsächlichen Verhaltens auswählen zu können, und einer „Arena“ welche es erlaubt die Reaktionen von Systemen auf Testfälle zu beobachten und diese datengetrieben vergleichen zu können. Wir nennen diese Plattform daher ein “Software Observatorium”, da diese einen Ort darstellt in der das Verhalten einer Vielzahl von Systemen beobachtet und verglichen werden kann.

Dedication

To my exceptional Onkel Roland who first introduced me to the fascinating universe of computers and the Internet ...

Acknowledgement

I would like to thank

... my supervisor Colin Atkinson for his excellent support, long-standing belief in our collaborative work, great optimism, and never-ending challenges.

... my co-referee Uwe Aßmann for his helpful feedback and new perspectives on this research work.

... my colleagues at the Chair of Software Engineering for open discussions, especially the code search subgroup and their preliminary work on merobase.

... my family, especially my ever-supporting wife, kids, parents and in-laws, sisters and brother, and my wonderful grandmother and grandmother in-law for their support.

... finally, all the Open Source communities for their ever-growing supply of great software products and repositories for free.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API Application Programming Interface
AST Abstract Syntax Tree
AUTG Automated Unit Test Generation

BC Branch Coverage

CDS Code-Driven (Code) Search
CUT Class Under Test

DAG Directed, Acyclic Graph
DSL Domain-Specific Language

EAV Entity-Attribute-Value Model
ECG Exhaustive Call Graph
ER Entity-Relationship (Model/Diagram)
ETL Extract, Transform, Load

IDCS Interface-Driven Code Search
IDE Integrated Development Environment
IoC Inversion of Control

JDK Java Development Kit

LOC Lines of Code
LSL LASSO Scripting Language
LQL LASSO Query Language

MS Mutation Score

NLP Natural Language Processing

OCG Observed Call Graph
OLAP Online Analytical Processing
OLTP Online Transaction Processing

RDBMS Relational Database Management System

SCG Syntactic Call Graph

SCM Source Code Management System
SDK Software Development Kit
SM Stimulus Matrix
SQL Structured Query Language
SRM Stimulus Response Matrix
SSN Sequence Sheet Notation
SUT System Under Test

TDS Test-Driven (Code) Search

UML Unified Modeling Language

Part I

Foundations

1 Introduction

1.1 Motivation

It has long been recognised that Open Source software repositories contain a goldmine of knowledge, experience and solution patterns that could help to dramatically reduce the costs of engineering high-quality software systems. Early attempts to exploit this resource focused on using traditional "logico-deductive" approaches of computer science [4], under the general banner of "software reuse" [149], to create easily-searchable indexes of software[1] and syntax-based platforms for analysing their contents [203, 212]. More recently, researchers have started to analyse software using the more statistical techniques of data science (e.g., natural language processing, machine learning and artificial intelligence) which have the added advantage of being optimised for "big data" and thus applicable at the scales of modern software repositories.

Since source code generally contains a mix of metadata elements (a.k.a., annotations or tags) and informal descriptive elements (e.g., comments) as well as formal, grammar-governed elements (i.e., the parsable instructions expressed in a programming language), it lends itself to analysis by the "semi-structured data processing" and "natural language processing" (NLP) techniques of data science. The "naturalness hypothesis" proposed by [4] also offers a strategy for applying these techniques to the formal elements of source code. This basically holds that since "software is a form of human communication", even the formal elements of code within software corpora "have similar statistical properties to natural language corpora" and thus are amenable to data science techniques.

Some researchers have proposed the term "big code" to capture the application of statistical "big data" technologies to software, often in tandem with the aforementioned "naturalness hypothesis". It also makes sense to extend this notion of "big code" to include approaches trying to apply "logico-deductive" approaches at a large scale, such as the BOA platform which has adapted traditional syntax analysis techniques to make them applicable in an "ultra-large", federated software repository [74, 75], and SOURCERERCC [209] for "scaling code clone detection to big code" (e.g., large repositories [164]).

[1] These are often called code search engines.

These existing "big code" technologies have a major weakness, however, which is their reliance on purely "static" analysis approaches that do not involve the execution of the software of interest. Instead, they use techniques like abstract interpretation [56] (i.e., via abstract models) to reason over all possible program states and paths. But the resulting imprecision severely limits the kind and quality of information that can be "known" or demonstrated with certainty about the dynamic properties of software (i.e., its behavioural semantics and non-functional execution properties). Such properties are among the most important attributes of software, and are what make it unique among the types of data to which big data techniques are applied.

"Logico-deductive" approaches, on the other hand, face a fundamental obstacle in the form of Rice's theorem which states that "all non-trivial semantic properties of programs are undecidable" [201]. So mathematically, any property that is considered non-trivial is (a) neither true for every partial computable function, nor (b) false for every partial computable function. This means that no general algorithm exists for deciding non-trivial questions about the behaviour (i.e., dynamic properties) of software (as a consequence of the unsolvable "Entscheidungsproblem" [242]). It follows that the only properties of software which can be automatically decided, in the general case, are static properties.

Although statistical techniques are not subject to Rice's theorem, they can only operate on the data they have available, which in the case of existing "big code" technologies is only static data about the software (often referred to as syntactic properties). It is certainly possible to make inferences about the run-time behaviour of software from static properties (e.g., syntactic properties such as identifiers, comments, annotations etc.), for example by applying the naturalness hypothesis, but the resulting inferences are inherently unreliable due to the absence of any required relationship between such elements and the "true" behaviour of a system. For example, although it is good practice to make the name of a class or method reflect its functionality (i.e., dynamic behaviour), this is not required.

The unavailability of dynamic data (i.e., data gained by executing software) in current "big data" technologies is a big weakness for two main reasons. First, it has long been known [76] that a combination of static and dynamic approaches presents the opportunity to develop more effective and also novel analysis and estimation approaches that provide better results than dynamically or statically derived data alone. The former tend to give overestimates since they consider all possible states (i.e., they are sound, but imprecise) while the latter give underestimates since only a subset of all possible concrete executions is considered (i.e., they are precise, but unsound) [76]. Second, statistical techniques are readily applicable to dynamically generated data, and are often drivers of the most efficient algorithms of a particular category (e.g., search-based testing [174] etc.). The problem is obtaining the

required dynamic data at the quantities needed to effectively complement the static data currently used in “big code”.

1.2 Problems

The full promise of “big code” will not be realised, therefore, until dynamic data is obtainable and analysable at scales commensurate to that currently possible with static data. Significantly scaling up the collection of dynamic (i.e., execution-based) data about software presents several significant challenges, however. This section outlines the key problems in the current state of the art.

P1. Manual Curation of Executable Software Corpora

Since the majority of modern software systems are encoded in object-oriented programming languages that require multiple program units (e.g., classes, interfaces) to be present and correctly interconnected (integrated) at run-time, the executability of a software system is not guaranteed, even if the constituent units are all compilable and fault free. Execution failures can occur at any time if any of the program units turn out to be missing, incompatible or wrongly combined.

Modern “build automation” tools and platforms such as Maven [233, 217] help address this problem by storing information (often called metadata) about the identities and relationships between the separate program units comprising software systems, but these can not guarantee the permanent presence or correctness of the units. Since online software repositories are usually subject to constant change and updates, inconsistencies and unavailabilities frequently occur. At the time of writing, therefore, the creation and management of repositories storing reliably executable “corpora” of software – frequently known as “corpora curation” – is a highly time-consuming and laborious process which has to be repeated on a frequent basis [68]. This, in turn, has the consequence that the resulting executable corpora created (e.g., for software engineering experiments) are typically small, rigid and brittle.

P2. Lack of Awareness of “True” Behaviour

Selecting software that possess certain properties is a basic prerequisite for big code. However, because the big code approaches mentioned previously exclusively use static techniques, they have no knowledge about the actual, “true” behaviour of a software system. In other words, they are only aware of the purported behaviour of the system (i.e., that implied by the textual identifiers (i.e., names) appearing in the

code) rather than the "true", de facto behaviour of the system, which results from its execution on a computing platform.

Except in extremely small cases, Rice's theorem means that the full, actual behaviour of a system can never be established automatically in practice, because it requires the responses of the system to be observed for all possible combinations of input parameters. However, by making observations of a system's responses when executed, it is possible to gain at least some partial knowledge about a system's true behaviour – an approach sometimes referred to as "behavioural sampling" [191]. Several research prototypes of code search engines have been developed which include support for behavioural sampling, such as MEROBASE [117], SOURCERER (CODEGENIE) [154, 17] and R6 [200], in order to increase the precision of semantic searches. These are variously referred to as "test-driven" or "test-case driven" [212, 119] code search engines. However, most of these are no longer available, and none are able to support behavioural sampling at the ultra-large scale needed for "big code".

P3. Statically-Defined System Boundaries

A key prerequisite for any software analysis approach is defining the scope of the measurements made on the software under consideration. As mentioned above, software systems are in general composed of multiple compilation units, which in turn, are composed of multiple elements at different levels of granularity (i.e., methods, statements etc.). Moreover, many of these elements may not have been written for the specific application of interest, but may belong to general libraries which are reused (i.e., invoked) by many different kinds of systems. There are therefore several possible criteria for defining the set of software elements to be included when analysing a system (i.e., when determining the boundary of the system under consideration), both in terms of purpose and form.

All existing big code techniques mentioned above make these boundary decisions using static approaches, which means they have no knowledge of the true behaviour of the system. For example, the scope of analysis could be limited to a single compilation unit, or to all the compilation units within a particular project or package etc. However, the fact that there are static dependency relationships between software elements is no guarantee that there are run-time dependencies between software elements. For example, a class which is statically imported by another class may not be referenced in any method call expressions in that class, so may not actually be called by it under any circumstances. It is important, therefore, that dynamic dependency information, as well as static information, is used to define the boundaries of software systems in analysis approaches.

The general area of research which deals with dynamic dependency and invocation analysis is referred to as program slicing [250], and many approaches have been proposed [254]. However, there is no unified model of how these tasks should be applied, and how behaviour observations can be used to define system boundaries. Moreover, all the existing approaches are research prototypes which are focused on small scale systems.

P4. Heterogeneous, Stimulus Definition Approaches

In order to obtain run-time observations of the behaviour of a software system, it is obviously necessary to execute it under controlled conditions. This activity is usually called testing, and the stimulus descriptions used to exercise the system are usually called tests. Testing is a major area of software engineering [6], and is supported by a large variety of dedicated technologies and tools. The most widely used approaches used to describe tests are the "xUnit" family of unit testing frameworks which are designed to complement the programming language used to write the software under test mainly using inheritance conventions or syntactic metadata such as annotations (e.g., JUnit for Java [132], PyUnit/unittest for Python [193] etc.).

Being able to write tests in a normal programming language, simply by following test-related conventions or adding annotations is a double-edged sword, however. On the one hand it means that the full power of the general purpose programming language in question can be used to write the tests, and only a minor number of new annotations have to be learned to describe how to interpret them. However, it also means that testers have a huge amount of freedom as to how they structure test software and how they apply the annotations. While this works when defining tests for single systems, it represents a major obstacle to the analysis of large numbers of software systems since there is little if any uniformity in the approaches used by different test writers. As a result, there is a huge amount of heterogeneity in the test definition approaches applied by different programmers even when using the same testing technology. When other testing tools and technologies are considered, the level of heterogeneity is even larger.

P5. Ad Hoc Behaviour Recording Approaches

As well as executing software under controlled conditions in order to obtain analysable observations, it is necessary to record the execution information in a systematic way. However, approaches used to do this are even more heterogeneous than those used to define the software tests, as discussed in *P4*. Moreover, it is not

only necessary to capture the system behaviour (i.e., observable outputs) it is also necessary to tie them explicitly to the tests (i.e., input parameters and systems states etc.) which lead to them. Unfortunately, the testing tools available today all use their own ad hoc, proprietary approaches for recording execution data and linking them to tests, and few if any of them are designed with ultra-large scalability in mind.

P6. Ad Hoc Process Definition and Management

As can be seen from *P1* to *P5*, many of the key technologies needed to support the ultra-large scale application of dynamic analysis approaches and the collection of true behavioural information either do not yet exist or use proprietary and/or ad hoc approaches. Moreover, even when they do exist, there is no well-defined tool or language for tying them together into a pipeline covering the whole process of dynamic software analysis. Researchers that have tried to do this in the past have therefore had to define their own ad hoc approaches using general purpose scripting languages and operating system commands. As well as being inefficient and error-prone, such ad hoc approaches do not lend themselves to scaling up to the levels required for big code and for defining repeatable experiments.

1.3 Requirements

To address the aforementioned problems and bridge the current gap in fulfilling the vision of big code, a dedicated platform for the ultra-large-scale collection and analysis of dynamic properties of software is needed, analogous to the BOA platform for the ultra-large-scale analysis of static software properties. Such a platform would essentially represent an *observatorium* (or observatory) for the large-scale observation of software system behaviour and application of dynamic analysis techniques. Such a platform would ideally need to meet the following key requirements.

R1. Automatically Curated Software Corpora

In the software engineering research community, the creation and ongoing management of software corpora for experimental analysis is generally referred to as "corpora curation", and is often performed manually, especially if executability is required. The first requirement the envisaged observatorium for big code needs to fulfil is the automatic curation of software corpora, so that users can be provided with data sets that are executable without huge amounts of manual effort. The

required automatic curation capability needs to be efficient at the ultra-large scale and accommodate ongoing additions and changes to corpora. This first requirement addresses the fundamental challenge of *P1*, where executable software corpora are typically curated manually.

R2. True-Behaviour-Aware Software Selection and Comparison

In order to perform specific experiments and focused analyses, users need to be able to select subsets of the software from the underlying curated repository that satisfy specific properties [251]. As mentioned above, current big data tools are only able to support software selection through static techniques. A key capability of the envisaged observatorium is to allow dynamically obtained data to be included in the selection process, and if desired, to be used exclusively. Although test-driven search engine prototypes such as MEROBASE etc., supported such a "behavioural sampling" capability, they only did so at a small scale, and with low recall. The envisaged observatorium needs to do so at an ultra-large scale and with a higher level of recall.

For many purposes, such as software experimentation, or creating a tool that solves a certain problem, it is desirable to not only establish whether software systems exhibit similar behavioural relationships (i.e., are functionally equivalent), it is also desirable to compare systems based on additional measurable engineering goals such as non-functional properties. It is therefore necessary to establish advanced discrimination criteria to further distinguish behaviourally similar systems. This ultimately leads to the requirement that a software observatorium needs to define, measure and apply advanced selection criteria for the purpose of characterising and discriminating between systems. For instance, implementational diversity of software systems with respect to their structural code design properties is one desirable property [179], since a set of software systems is typically required to exhibit a certain level of diversity.

This second requirement addresses the fundamental challenge of *P1* where executable corpora are a prerequisite for true behaviour-aware software selection, and it addresses *P2*, the problem that current approaches are unaware of true behaviour.

R3. True-Behaviour-Aware System Boundary Models

A key question when obtaining metrics on software (whether dynamic or static metrics) is to define the extent of the software. As mentioned above, existing big data approaches only use statically derivable metrics to make this determination, but many of the "natural" scoping criteria users might want to apply inherently

involve dynamic information as well. For example, a natural criteria would be to include all the code which is involved in delivering the behaviour offered through a particular interface. However, existing approaches are not only unable to support such true-behaviour-aware boundary defining criteria, but lack even the concepts and vocabulary to describe them. The envisaged observatorium needs to support such a capability in a way that is efficient at a large scale. The third requirement addresses the limitations arising from *P3*.

R4. Unified Stimulus/Response Data Structure and Analysis Platform

As explained in the previous section, a major obstacle to large-scale dynamic software analysis at the present time is the large variety of heterogeneous and disconnected languages/tools for defining software stimuli (i.e., tests), executing them on multiple software systems, recording the responses and extracting useful information from the results. To the extent that these do exist, existing realisations of these tools are disjoint and unscalable. The envisaged observatorium therefore needs to provide a unified approach to the stimulation and observation of software which employs a unified data structure and associated conceptual model across the whole stimulus/response process, and provide a platform that is able to efficiently generate, store and analyse observation data at an ultra-large scale. This requirement addresses problems *P4* and *P5*.

R5. Dedicated Pipeline Definition Language

Finally, all the individual realisations of the aforementioned requirements need to be woven together into the envisaged platform to provide users with a unified view of the whole data creation and analysis pipeline. This in turn will require an overarching domain-specific language for dynamic software analysis which allows users to define process pipelines that apply all the steps involved, using the aforementioned capabilities, in an intuitive, seamless and unified way. And of course the enactment of such pipelines needs to be scalable to the ultra-large level. This requirement addresses *P6*.

1.4 Hypotheses and Research Method

The premise underlying this thesis is that a tool, scalably supporting the requirements outlined in *R1* through *R5* – a “Software Observatorium” – can be constructed using

today's technologies to support and promote the big code vision outlined above. In other words, the research presented in this thesis is based on the following three hypotheses —

Hypothesis 1. *It is feasible to build a software observatorium that scalably and efficiently supports requirements R1 through R5, in a user-friendly way, to support the dynamic analysis and comparison of software systems.*

Hypothesis 2. *It is feasible to use such a software observatorium to build new and/or improved software engineering tools.*

Hypothesis 3. *It is feasible to use the software observatorium to support better evaluations of software engineering tools and approaches.*

The research approach used to explore the validity of this hypothesis is the "design science" methodology. To this end, we therefore applied the seven design science ingredients of Hevner et al. [111, 169] in the following way —

- *Problem Relevance*: we identified the need for, and the relevance of, the envisaged software observatorium (Chapter 2)
- *Design as an Artefact*: we constructed a prototype platform – LASSO, the Large-Scale Software Observatorium (Chapter 12)
- *Design Evaluation*: we evaluated the utility, quality, and efficacy of the developed technology by using it to (a) construct new and improved software engineering tools (Chapter 13 and 14), and (b) perform experiments to evaluate software engineering tools (Chapter 15)
- *Research Contributions*: we presented various significant contributions of the approach to big code and software engineering in general (Chapter 17)
- *Research Rigor*: we developed formal models for the novel data structures and techniques used in the observatorium (Chapter 4 and Chapter 6), created a concrete DSL for effectively using them (Chapter 10), and applied rigorous statistical methods to evaluate them (Chapter 15)
- *Design as a Search Process*: we constructed and evaluated the prototype in a cyclic manner using the latest state-of-the-practice software engineering technologies (Part II and III)
- *Communication of Research*: we published key aspects of the technology in various international workshops, conferences and journals (Section 1.5).

1.5 Research Communication

The research conducted as part of this thesis has been presented in various international workshops, conferences and journals, including —

1. C. Atkinson *et al.*, "On the synergy between search-based and search-driven software engineering," in *Search Based Software Engineering*, G. Ruhe and Y. Zhang, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 239–244

2. M. Kessel and C. Atkinson, "Measuring the superfluous functionality in software components," in *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, ser. CBSE '15, Montréal, QC, Canada: Association for Computing Machinery, 2015, 11–20

3. M. Kessel and C. Atkinson, "Ranking software components for pragmatic reuse," in *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*, 2015, pp. 63–66

4. M. Kessel and C. Atkinson, "Ranking software components for reuse based on non-functional properties," *Information Systems Frontiers*, vol. 18, no. 5, pp. 825–853, 2016

5. M. Kessel and C. Atkinson, "Integrating reuse into the rapid, continuous software engineering cycle through test-driven search," in *2018 IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering*, 2018, pp. 8–11

6. M. Kessel and C. Atkinson, "A platform for diversity-driven test amplification," in *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2019, Tallinn, Estonia: Association for Computing Machinery, 2019, 35–41

7. M. Kessel and C. Atkinson, "Automatically curated data sets," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019, pp. 56–61

8. M. Kessel and C. Atkinson, "On the efficacy of dynamic behavior comparison for judging functional equivalence," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019, pp. 193–203

9. M. Kessel and C. Atkinson, "Diversity-driven unit test generation," *Journal of Systems and Software*, vol. 193, 2022

For practitioners and researchers, the prototype platform LASSO and its artefacts have been made publicly available with the aim of establishing a long-term community of users. The LASSO project is available online —

- Official Website: `https://softwareobservatorium.github.io/`
- Software Repository: `https://github.com/SoftwareObservatorium/`

1.6 Outline

The thesis has six parts. Part I provides the foundations for the presented research by outlining its goals and foci. Chapter 1 starts with a description of the addressed requirements, the investigated hypotheses and the applied methodology, while Chapter 2 presents two motivating examples to illustrate the need for the envisaged software observatorium and the services it offers to users. Chapter 3 then clarifies the terminology and formal model used in the approach.

Part II introduces the arena component of the software observatorium that facilitates observations and comparisons of large numbers of software systems. First, Chapter 4 presents the notion of sequence sheets as a unified way of representing stimuli of, and responses from, software systems, while Chapter 5 describes how custom scopes for measuring software metrics can be defined using dynamic behaviour information. Chapter 6 then proposes a unified matrix data structure to support large-scale software observation and analysis of multiple systems and test sequences.

Part III presents the technologies that support the population of the observatorium with appropriate stimulus sheets and software systems. Chapter 7 first introduces the underlying executable software corpus that provides the foundation for the automated curation of software systems. Chapter 8 then presents the text-based (NLP-driven) search technology used to retrieve software systems based on their textual properties, while Chapter 9 describes the test-driven approach for filtering the retrieved candidates through behaviour sampling.

Part IV explains how the observatorium components from Part II and Part III are integrated through a dedicated, domain-specific pipeline language that allows users to write highly compact, flexible and reusable data collection and analysis scripts. Chapter 10 presents the language's features for supporting efficient arena population and execution pipelines while Chapter 11 describes how the collected data can be used and analysed in mainstream big data analytics tools.

Part V evaluates the developed technology by describing a prototype observatorium implementation and showing how it can be used to support advanced software

tools and experimentation. Chapter 12 first describes the prototype observatorium, LASSO, before Chapters 13 and 14 demonstrate its application in two distinct software engineering domains. The former presents two services that focus on searching for, and curating, software systems with specific properties, while the latter presents new approaches for generating and amplifying tests by exploiting software diversity. Chapter 15 then shows how LASSO can be used to support software experimentation.

Finally, Part VI concludes the thesis by placing it in the context of the state of the art and summarising its contributions. Chapter 16 starts by presenting related work, while chapter 17 concludes with a discussion of the validity of the hypotheses, the potential and limitations of the developed technology and possible directions for future research.

The main body of the thesis is complemented by an appendix which lists all the LSL scripts used to implement the discussed approaches.

2 Motivational Examples

This chapter presents some example scenarios to motivate the need for, and showcase the features of, the software observatorium developed in this thesis and to serve as the basis for the running examples. Two concrete use cases are presented which involve the need to combine statically and dynamically obtained data at a potentially large scale, accompanied by a general discussion of the different kinds of use cases for which the observatorium's capabilities are needed.

2.1 Example 1 - Test Set Quality Assessment

This example deals with the question of determining the quality of a set of tests for a particular piece of software using well-known metrics (Section 16.4.2). For concreteness and simplicity, we focus on an implementation of the well-known *stack* data abstraction [147]. More specifically, we assume the goal is to answer the following question for a set of tests, T, that have been written to test a stack implementation, s —

Question 1. What is the quality of T, for the stack implementation s, measured in terms of MS (Mutation Score) and BC (Branch Coverage)?

This is a core question arising in software engineering projects, and is a capability built into many software testing environments. However, these capabilities are invariably highly opaque in terms of what process and assumptions they apply, and highly inflexible in terms of giving users the ability to change the process and assumptions.

To make the example as concrete as possible, Listing 1 shows the stack implementation, s, while Listing 2 shows a JUnit representation of one of the tests, t_i, of s in the set T. JUnit [132] is the most widely used way of defining tests for Java software. Although we only show one test for illustration purposes, we assume there are y tests in total.

As can be seen in Listing 2, a test is an executable piece of software which contains a sequence of invocations of methods of an instance of class s, with specific input parameters, and checks whether the outputs from one or more of the method invocations have particular values (i.e., through the assertion statements).

```java
import java.util.ArrayList;

public class ArrayStack {
    private ArrayList list = new ArrayList();

    public Object push(Object e) {
        list.add(e);
        return e;
    }

    public Object pop() {
        return list.remove(list.size() - 1);
    }

    public Object peek() {
        return list.get(list.size() - 1);
    }

    public int size() {
        return list.size();
    }
}
```

List. 1: Java Implementation of Stack s

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class ArrayStackTest {

    ...

    @Test
    public void test_ti()  throws Throwable  {
        ArrayStack arrayStack = new ArrayStack();
        arrayStack.push("hello world");
        int int0 = arrayStack.size();
        assertEquals(1, int0);
    }

    ...
}
```

List. 2: JUNIT Method Representation of a Test, $t_i \in T$

```
1  import java.util.ArrayList;
2
3  public class ArrayStack {
4      private ArrayList list = new ArrayList();
5
6      public Object push(Object e) {
7          list.add(e);
8      return e;
9      }
10
11     public Object pop() {
12         return list.remove(list.size() - 1);
13     }
14
15     public Object peek() {
16         return list.get(list.size() - 1);
17     }
18
19     public int size() {
20         //return list.size();
21         return 0; // mutated location
22     }
23 }
```

List. 3: A Mutant, $m_j \in M$, for Implementation of s

2.1.1 Arena Setup

The measurement of MS and BC presents various challenges. In the case of BC, it is necessary to execute each test $t \in T$ on s and compare static information about the number of paths in s with dynamic information about the path taken by each execution in order to establish a coverage value for T. In the case of MS, however, it is not only necessary to execute each test in T on s, each test must also be executed on a set of mutants, M, obtained from s by applying carefully chosen mutation operators. Assuming that there are $|M|$ mutants, this gives a total of $x = |M| + 1$ implementations (i.e., number of mutant implementations, M, plus implementation s), resulting in a total of $x \times y$ test invocations. For each case, it is necessary to store information about the execution path taken by s (to determine BC) and the outputs returned by s (to determine which mutants are killed). Listing 3 shows an example of a mutant, $m_j \in M$, obtained from s by applying the "primitive returns" mutation operator [52] that substitutes an existing return value with 0 (the mutated statement is Line 21).

The component of the observatorium that supports the execution of such constellations of tests and implementations in an efficient, understandable and analysable way is the so-called "arena". This is essentially the heart of the observatorium and is conceptually the place where large numbers of invocations of related software systems are performed in a carefully controlled, organised and easily accessible way.

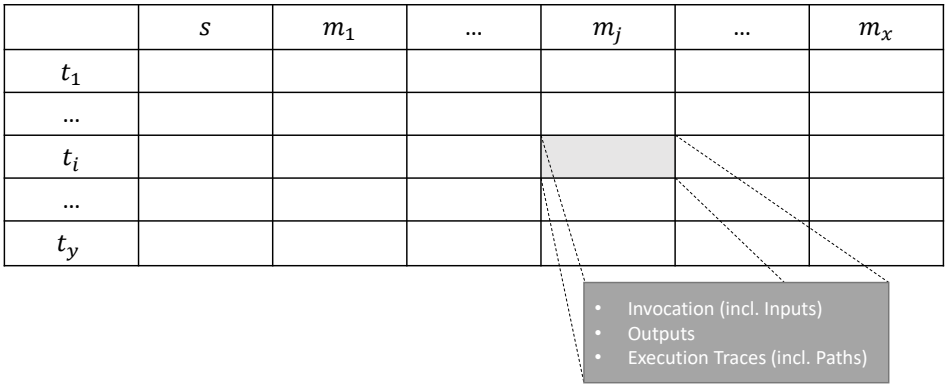


Fig. 2.1.: Arena Setup - Example 1

The basic arena setup for this example is shown in Figure 2.1. This shows the various implementations (in this case s and its mutants M) which are the subject of testing organised in one dimension (i.e., the x dimension or columns), and the tests which stimulate these implementations in the other dimension (i.e., the y dimension or rows). The job of the arena is to apply all tests to all implementations and to store information about the results. As shown in Figure 2.1, for each test/implementation pair (corresponding to a cell) this information includes the test inputs used in the invocation, the outputs returned by the implementation and a trace of the execution path followed in the execution.

This two-dimensional constellation of test/implementation pairs is manifested and stored in two basic forms by the observatorium. The first is as SMs (stimulus matrices) which describe the participants (i.e., tests and implementations) and their two-dimensional arrangement in a way that sets up the arena and describe all the executions that need to take place (Section 6.1). The second is as SRMs (stimulus response matrices) which augment SMs with information about the results of the executions (Section 6.2). In a sense, the former describes the inputs to the arena, in order to set it up for the multiple executions, while the latter describes the resulting outputs and observed behaviour for these inputs.

2.1.2 Sequence Sheets

Although powerful, the JUNIT approach to writing tests does not lend itself to describing the aforementioned information in a sufficiently abstract and concise way. This is important because any accidental (i.e., unnecessary) code is multiplied by the huge number of test executions that need to be described and recorded. The observatorium therefore offers a specially designed and intuitive notation – sequence

	A	B	C	D
1		create	?stack	
2		push	A1	?p
3		size	A1	

Fig. 2.2.: Sequence Sheet Body for Test Method $t_i \in T$

```
1  Stack {
2      Stack() // empty (default) constructor
3      push(Object)->Object
4      pop()->Object
5      peek()->Object
6      size()->int
7  }
```

List. 4: Interface of Stack Abstraction

sheets – for representing test methods, their application to implementations and the resulting behaviour. Sequence sheets abstract away from the low-level details in normal code to focus on the essential invocations of the methods of the executed implementations, which share the same logical interface (Section 4.2).

Figure 2.2 shows the sequence sheet representation of a test method for $t_i \in T$, which corresponds to the JUNIT version shown in Listing 2. This is written against the abstract interface supported by s, and all other implementations in the arena, shown in Listing 4.

The sequence sheet in Figure 2.2 essentially represents the body of the test method for t_i. This method also has a normal method signature which includes the subject of a particular implementation as well as any further parameters that can be varied. Note that expressions like `?p` starting with a question mark, depict input parameters. In this case, the implementation passed to the sequence sheet (i.e., class `ArrayStack`) is passed via the `?stack` parameter, whereas the value pushed onto the stack (i.e., the string `'hello world'`) is passed as input parameter `?p`. The `create` method in cell *B2* of the sequence sheet is a special method that is used to create an object (i.e., instance) of the passed stack class. The corresponding method signature of the sequence sheet has the form – `TestTi(stack:Stack,p:Object)`, and its invocation on the actual parameters from test t_i is represented as `TestTi('ArrayStack','hello world')`.

Figure 2.3 schematically shows the form of the SM which describes the exact conditions under which each implementation is executed by each test in order to create the required observations. This shows the method invocation required in each cell in a black box notation (i.e., using the traditional method invocation syntax from programming languages). However, the observatorium also allows the method

	s	m_1	...	m_j	...	m_x
t_1				TestT1(m_j, ...)		
...				...		
t_i	TestTi(s, ...)	TestTi(m_1, ...)	...	TestTi(m_j, ...)	...	TestTi(m_x, ...)
...				...		
t_y				TestTy(m_j, ...)		

Fig. 2.3.: Stimulus Matrix - Example 1

	A	B	C	D
1	«obj»	create	?stack	
2	'hello world'	push	A1	'hello world'
3	1	size	A1	

Fig. 2.4.: Sequence Sheet Representation of the Result of the Execution `TestTi(`m_j`, ...)`

invocation to be shown in a white box style in which the sequence sheet body is shown with the actual parameter values.

After performing all the executions defined by the stimulus matrix in Figure 2.3, the arena creates a corresponding SRM which includes the additional run-time results. At the black box level, the SRM looks identical to the SM from which it was created, but at the white box level, the results of each execution can be seen in the form of an actuation sheet.

Figure 2.4 shows the results of applying Test t_i to m_j through the invocation of `TestTi(`m_j`, ...)` in expanded white box notation. This is similar to Figure 2.2 except that the actual returned values and other outputs derived in that execution are explicitly included. The observatorium allows SRMs to be stored in any convenient format such as CSV files which can be imported by tools like R [239] for data-driven analysis (Section 6.7).

2.1.3 Pipeline Script

Even this small example involves a number of steps which the observatorium has to be instructed to perform —

1. creation of the mutants from s,
2. definition of the test methods T,
3. definition of the SM,

4. execution of the contained invocations,

5. calculation of the metrics,

6. saving of the resulting SRM.

To facilitate the description of such processes, which we call pipelines, the observatorium offers a dedicated scripting language, LASSO Scripting Language (LSL). The LSL script to define the pipeline needed to address the question in this example is shown in Listing 5. Note that the data-driven analysis of the collected measures is performed in large-scale data analytics (or statistical) tools that are well-integrated into the prototype platform that realises the observatorium (Section 6.7).

2.2 Example 2 - Heteromorphic Redundancy Assessment

This example deals with the question of establishing the level of redundancy of heteromorphic implementations of a piece of functionality (i.e., functional abstraction) within a software repository. Heteromorphic redundancy occurs when there are multiple, diverse (i.e., none clone) implementations of that functional abstraction within the repository in question (Section 14.1). For concreteness and simplicity, we focus on the Base64 functionality used to encode and decode information transmitted over the Internet (see RFC 4648 [130]), and the well-known Maven Central repository (Section 12.4.1). The goal, therefore, is to answer the following question —

Question 2. What is the level of heteromorphic redundancy of implementations of Base64 functionality in the Maven Central repository?

This can be interpreted as "what is the number of diverse (i.e., none clone) implementations of Base64 in Maven Central?". The key, implied, non-functional requirement is to address this question with maximum precision and recall (cf. performance metrics [168]). In other words, the result should **only** contain diverse implementations of Base64 (minimal number of false positives) and should contain **all** implementations of Base64 (minimal number of false negatives relative to the contents of the repository).

This example is more challenging than the first example in several ways. First, in order to judge whether different implementations of a functional abstraction such as Base64 are distinct, it is necessary to measure important metrics on the implementations such as lines of code (LOC) and cyclomatic complexity (e.g.,

```
dataSource 'mavenCentral2020' // select from given data source
/* define new analysis pipeline */
study(name:'Stack-Test-Quality') {
    /* selects a given stack implementation s */
    action(name:'select', type:'Select') {
        abstraction('Stack') {
            queryForClasses "*:*"
            filter 'name:"ArrayStack"' // assumes some existing Java class (dummy)
        }
    }
    /* defines an execution profile for the arena */
    profile('myProfile') {
        scope('class') { type = 'class' } // measurement scope
        environment('java8') { // execution environment
            image = 'maven:3.5.4-jdk-8-alpine' // (docker) image template
        }
    }
    /* populate and execute the arena */
    action(name:'execute',type:'ArenaExecute') {
        sequences = [ // defines sequence sheets using LSL keywords
                'ti': sheet(stack:'Stack', p:'hello world') {
                    row '',  'create', '?stack'
                    row '',  'push',   'A1',     '?p'
                    row '',  'size',   'A1'
                },
                ... // other tests
        ]

        dependsOn 'select'
        includeAbstractions 'Stack' // select implementation from former action
        profile('myProfile')
    }
    /* measure MS */
    action(name:"mutationScore",type:'Pitest') {
        dependsOn "execute"
        includeAbstractions 'Stack'
        profile('myProfile')
    }
    /* measure BC */
    action(name:"branchCoverage",type:'JaCoCo') {
        dependsOn "execute"
        includeAbstractions 'Stack'
        profile('myProfile')
    }
    /* analyse obtained measures within LSL (optionally, export) */
    action(name:'analyse') {
        dependsOn 'branchCoverage'
        includeAbstractions 'Stack'
        // custom analysis based on SRM structure
        execute() {
            def stack = abstractions['Stack']
            def branchTotal = srm(abstraction: stack)
                    .systems['ArrayStack'].observations['cc.branch.total']
            def mutationScore = srm(abstraction: stack)
                    .systems['ArrayStack'].observations['mutation.score']
            ... // do something
        }
    }
}
```

List. 5: Pipeline - LSL Script for Test Set Quality Assessment

McCabe [173]) etc. This is not a problem when all the software that implements the functional abstraction is clearly contained within one class, such as with the stack in the first example. However, in general, the code that participates in delivering the functionality in question extends over multiple classes and is mixed up with code that is not involved in delivering that functionality. So how is the "scope" of the implementation defined, i.e., how is the code that is "part of" the implementation distinguished from the code that is not? This question cannot be resolved by static analysis techniques alone (Chapter 5).

Second, this example involves more complicated processes and criteria for populating the arena with implementations including harvesting them automatically from the repository (i.e., Maven Central in this case). To avoid wasting time and resources, the arena needs to be populated with implementations which are reasonably likely to be implementations of the required Base64 functionality. Note that it is not possible to prove, analytically, that software delivers specific functionality due to Rice's theorem [201]. Confidence in the functional similarity of different implementations can therefore only be obtained by observing their execution under identical conditions (Section 16.1).

Third, this example also involves more complicated processes and criteria for populating the arena with tests. The purpose of the tests it to produce information that (a) increases confidence that the implementations are functionally equivalent, and (b) provides evidence of which code is involved in delivering the functionality of interest (i.e., Base64). Both of these criteria require a set of tests of the highest possible quality, ideally generated automatically.

2.2.1 Implementation Harvesting

To facilitate the implementation discovery step of the arena population process in cases like this, the observatorium provides a powerful software search capability akin to mainstream code search engines. At its core is an index of the repository in question constructed using the SOLR/LUCENE full-text indexing engine [234]. This supports various text-based, NLP-driven retrieval capabilities offered in the form of a query language (e.g., interface-driven search in Section 8.3). For this example, an appropriate query for harvesting implementations of the Base64 encoding functionality (i.e., encoding data into the Base64 representation) would be —

```
Base64 { encode(byte[])->String }
```

and an appropriate query for harvesting implementations of the Base64 decoding functionality would be —

```
1  import org.junit.Test;
2  import static org.junit.Assert.*;
3
4  public class Base64Test {
5
6      ...
7
8      @Test // verifies class case (no "padding")
9      public void testEncode() { //
10         Base64 cut = new Base64();
11         byte[] actual1 = cut.encode("user:pass".getBytes());
12         assertEquals("dXNlcjpwYXNz", new String(actual1));
13     }
14
15     @Test // verifies if "padding" character '=' is present
16     public void testEncode_padding() {
17         Base64 cut = new Base64();
18         byte[] actual2 = cut.encode("Hello World".getBytes());
19         assertEquals("SGVsbG8gV29ybGQ=", new String(actual2));
20     }
21
22     ...
23 }
```

List. 6: Characterising Test Set for Base64

```
Base64 { decode(String)->byte[] }
```

These queries will return implementations that have a reasonable likelihood, on a textual basis, of being Base64 implementations (i.e., based on their class and methods names). However, in order to increase precision (i.e., reject false positives), the observatorium also supports behaviour sampling (a.k.a., test-driven search) which automatically applies a set of "characterising tests" to candidates, and rejects those implementations that fail (Section 9.1). These tests can be represented in JUNIT or the sequence sheet notation described above.

Listing 6 shows a small characterising test set for Base64 encoding. These can be defined by hand by a software engineer or generated automatically from a reference implementation of Base64 if one exists, such as that shown in Listing 7. When a reference implementation is used as the basis for a test-driven search it is also sometimes called a code-driven search (Section 9.2).

While test-driven search can significantly enhance the precision of the overall search (or harvesting) process, it can also dramatically reduce its recall if the set of returned implementations is limited to those that are immediately callable by the characterising tests (i.e., that directly satisfy the desired interface). To increase the recall, therefore, and return the largest possible set of implementations that theoretically implement Base64, the observatorium offers sophisticated adaptation capabilities to adapt implementations with different interfaces to that required (Section 9.3).

```
1  public class Base64 {
2      ...
3
4      public String encode(byte[] bytes) {
5          return java.util.Base64.getEncoder().encodeToString(bytes);
6      }
7
8      ...
9  }
```

List. 7: Reference Implementation for Base64

	A	B	C	D
1		create	?base64	
2		encode	A1	[]

Fig. 2.5.: Randomly Generated Test by EvoSuite (Represented as a Sequence Sheet)

As a final filtering step, since this example is interested in heteromorphic implementations of the functional abstraction in question, statically recognisable identical clones are removed using established clone detection techniques, supported by tools such as NiCad [54].

By combining these various technologies, the observatorium is able to offer an effective retrieval capability from an indexed repository that can return executable implementations of the functionality of interest with high precision and recall.

2.2.2 Test Generation

To facilitate the testing part of the arena population process in scenarios like this, the observatorium offers various automated test generation capabilities using established test generation tools. If a reference implementation is available, a white-box, automated unit test generation tool like EvoSuite [84] can be used to generate tests based on that implementation (Section 16.4.2). If not, then a tool with random test generation capabilities like Randoop [184] can be used.

Figure 2.5 shows a sequence sheet representation of a test automatically generated by EvoSuite for the reference implementation shown in Listing 7. EvoSuite generates JUnit tests, but the observatorium can translate these into sequence sheets. Note that in this case, the tests are fairly simple, involving only one method invocation to create an instance of the class and one method invocation for Base64 encoding.

	r	h_1	...	...	...	h_x
t_1	TestT1(r, ...)	TestT1(h_1, ...)				TestT1(h_x, ...)
...						
...						
...						
t_y	TestTy(r, ...)	TestTy(h_1, ...)				TestTy(h_x, ...)

Fig. 2.6.: Stimulus Matrix - Example 2

	A	B	C	D
1	«obj»	create	?base64	
2	'SGVs...GQ='	encode	A1	[72,101,...,100]

Fig. 2.7.: Response Sheet for the Second Example Test in Listing 6

2.2.3 Arena Setup

Once the required implementations have been harvested and the required tests have been generated, they can be used to populate the arena, similarly to the previous example. Figure 2.6 illustrates the general setup of the arena for this example.

The columns are populated by the different harvested implementations of the functionality of interest (h_1 to h_x) as well as the reference implementation r. The rows, on the other hand, are populated by the tests, ideally generated automatically from the reference implementation.

Once the SM has been defined, the arena can execute the defined tests and produce a corresponding SRM. Figure 2.7 illustrates the kind of result sequence sheet produced by applying the second test in Listing 6 to one of the implementations of the Base64 encoding abstraction.

2.2.4 Similarity Measurement

In order to judge how diverse alternative implementations of a functional abstraction are it is necessary to define some kind of similarity measure based on simple metrics like cyclomatic complexity or lines of code. However, this is challenging because the alternative implementations may be spread over multiple classes, and may include code that is not involved in implementing the desired functionality. The observatorium therefore provides technology to (a) help determine which code appears likely to be involved in the implementation of the functionality of interest, and to (b) help define the scope of the implementation. Both of these require

dynamic information from the execution traces of each of the invocations in the resulting SRM.

To determine the involved code we instrument the executed class implementations and trace their (transitive) method invocations starting from their "entry" methods that "match" the interface of the Base64 abstraction at hand (i.e., `encode(byte[])->String`). To define the desired scope, the maximum depth of the method call graph can be provided or white lists can be maintained in order to ignore methods and code elements from classes that are not of interest, for instance (Section 5.2).

Once the scope and involved code have been determined, the metrics needed to calculate the similarity measure can be determined statically and the pairwise similarity measures established. These can then be used to create the final result by rejecting alternative implementations which (a) are shown to not implement the functionality of interest because they fail one or more of the tests, and/or (b) are deemed to be too similar to one or more other implementations to be considered heteromorphically distinct.

The judgment of whether a harvested implementation "fails" a test can be made in one of three ways, based on what is chosen as the oracle —

1. Human engineers can play the role of the oracle, as is often the case in practice, by deciding what the correct result for a test should be. The work involved in this case can be minimised (thanks to the SRM) by using a discrepancy based approach in which humans only need to arbitrate between disagreements – that is, when different implementations give different results for the same test.

2. The reference implementation can be regarded as the oracle. In this case, harvested implementations are deemed to have passed a test if they deliver the same results as the reference implementation.

3. An automated voting system can be used to arbitrate disagreements between the alternative implementations. This is a generalisation of the first approach where arbitration is done automatically rather than manually by a human.

2.2.5 Pipeline Script

Not surprisingly, given the extra steps and technologies involved in this example, the LSL script defining the required pipeline is more complex. A possible LSL script for this example is shown in Listing 8.

```
dataSource 'mavenCentral2020' // select from given data source
/* define new analysis pipeline */
study(name:'Base64-Heteromorphic-Redundancy') {
    /* query Base64 implementations by interface signatures */
    action(name:'select', type:'Select') {
        abstraction('Base64') { // interface-driven search
            queryForClasses 'Base64{encode(byte[])->byte[]}'
            rows = 1000 // no. of Java classes to return
        }
    }
    /* reject code clones */
    action(name: "clones", type: 'Nicad6') {
        cloneType = "type2" // clone type to reject
        collapseClones = true // remove clone implementations

        dependsOn "select"
        includeAbstractions '*'
    }
    /* defines an execution profile for the arena */
    profile('myTdsProfile') {
        scope('class') { type = 'class' }
        environment('java8') {
            image = 'maven:3.5.4-jdk-8' // (docker) image template
        }
    }
    /* populate and execute the arena */
    action(name:'filter',type:'ArenaExecute') {
        sequences = [
                'testEncode': sheet(base64:'Base64', p2:"user:pass".getBytes()) {
                    row  '',    'create', '?base64'
                    row 'dXNlcjpwYXNz'.getBytes(),  'encode',   'A1',     '?p2'
                },
                'testEncode_padding': sheet(base64:'Base64', p2:"Hello World".getBytes()) {
                    row  '',    'create', '?base64'
                    row 'SGVsbG8gV29ybGQ='.getBytes(),  'encode',   'A1',     '?p2'
                }
        ]
        maxAdaptations = 1 // how many adaptations to try

        dependsOn 'clones'
        includeAbstractions 'Base64' // select implementations from former action
        profile('myTdsProfile')
        // match implementations and compute simple statistics
        whenAbstractionsReady() {
            def base64 = abstractions['Base64']
            def base64Srm = srm(abstraction: base64)
            // define oracle based on expected responses in sequences
            def expectedBehaviour = toOracle(srm(abstraction: base64).sequences)
            // alternatively, use any system as a (pseudo) oracle
            def referenceImpl = toOracle(srm(abstraction: base64).systems.first())
            // returns a filtered SRM
            def matchesSrm = srm(abstraction: base64)
                    .systems // select all systems
                    .equalTo(expectedBehaviour) // functionally equivalent

            // get LOC measures for advanced heteromorphic redundancy assessment
            def loc = matchesSrm
                    .systems.observations['cc.loc']
            // average
            double locAvg = loc.mean()
            log("Average number of lines of code is ${locAvg}")
        }
    }
}
```

List. 8: Pipeline - LSL Script for Heteromorphic Redundancy Assessment

2.3 Ultra-Large Scale Software Observation

The two examples presented above both involve the creation of one SRM. Although the size of SRMs is unlimited in both dimensions, in practice they rarely expand to a size that can be regarded as "big", in the sense of big code, or "ultra-large" in the sense of BOA [75]. However, applications of the observatorium can quickly expand to this size when multiple SRMs are involved, which is often, if not usually, the case.

The most straightforward scenario in which multiple SRMs become involved is when the observatorium is used to conduct an experiment (Chapter 15). For example, if the tests in Example 1 were generated by a test generation tool, such as EVOSUITE, the SRM setup in study one (see Listing 5) could be used to experimentally evaluate the performance of that tool by essentially repeating the scenario for a large number of alternative functional abstractions. If these are selected in a way that makes them a representative sample of the general population of interest (e.g., by random selection), the results obtained for each sample can be combined to obtain generalisable observations about the properties of the tool under study. Similarly, the setup used for Example 2 could be generalised to perform a study of the average level of heteromorphic redundancy in software repositories.

It is not only the need to create and process multiple SRMs over multiple functional abstractions that increases the amount of dynamic information to be created, stored, managed and analysed to the "big code" or "ultra-large" scale, it is also the need to repeat observation multiple times when randomised algorithms are involved. This is increasingly the case as tools become more sophisticated. The observatorium is therefore designed to handle multiple SRMs, which can be thought of as occupying a multiple-dimensional space. It also provides a flexible random sampling mechanism to make results more generalisable.

3 Formal Model and Terminology

This chapter presents the formal foundations of our approach to the large-scale dynamic analysis of software as well as the terminology used in the rest of the thesis. Our formal framework is loosely inspired by Gourlay's framework for formal foundations in software testing [97], with extensions for oracles from Staats et al. [218], and Barr et al.'s stimulus response model [24] which formalises the process of software testing through dynamic analysis.

3.1 Basic Object-Oriented Notions

We assume that software systems are constructed using object-oriented programming concepts. For our purposes the most important and fundamental of these are methods (also known as procedures), method invocations (also often referred to as method calls), classes, interfaces, objects and types.

3.1.1 Methods and Method Invocations

A **method** is an executable unit of functionality that can be invoked. A method is composed of two parts – a method signature and a method body.

A **method signature** describes the externally visible interface to a method that characterises how it should be invoked. It is composed of a name and two ordered lists of formal parameters – an ordered list of input parameters and an ordered list of output parameters. A formal parameter is composed of a name and a type. Note that the explicit object identifier used in the method invocation syntax of many object-oriented programming languages (e.g., Java) is regarded as one of possibly many input parameters, and the returned value, if one exists, is regarded as one of possibly many output parameters. Many object-oriented programming languages such as Java only allow one returned value. Based on mathematical logic and universal algebra for terms [114], a method signature for method m is formally depicted by its type-signature, $m : [I_1, ..., I_k] \rightarrow [O_1, ..., O_l]$, where $[I_1, ..., I_k]$ is the list of formal input parameter types, and $[O_1, ..., O_l]$ is the list of formal output parameter types.

Each of these parameters can be accessed either via its position or its name through indices.

A **method body** describes the executable behaviour of a method based on the information passed to it in the way prescribed by its signature. The body is said to "conform" to the signature. It uses lower-level operations (e.g., statements) to describe the method's executable functionality, which involves invocations of other methods.

A **method invocation** represents the invocation of (i.e., a call to) a method via its signature. Method invocations occur in the body of methods and have two ordered lists of actual parameters (which may be empty) – an ordered list of input parameters and an ordered list of output parameters. The form of the actual parameters has to match the method signature. Actual parameters can be literal values, references to objects, or fields for storing the values of output parameters. The number and types of the actual parameters in a method invocation must match the number and types of the formal parameters in the method signature. Note that for the sake of formalisation, we strictly assume a fixed list of positional parameters. Nevertheless, allowing a variable number of arguments to a method (e.g., Java's *varargs* or Python's *kwargs/args*), or allowing keyword-identified parameters (i.e., addressed by the parameter's name) that can be specified in an arbitrary order, are special cases which can be easily incorporated into the model.

Finally, since we are assuming the imperative programming paradigm, the **control flow** of operations inside a method body is determined by conditional statements, such as if-then-else statements and loops. These determine which branches of a method body are executed as a consequence of the actual parameters passed at the method's invocation. In our formal model, control flow concepts like conditionals and looping are also modelled as operations.

3.1.2 Classes and Objects

A **class** is an executable package of behaviour that is composed of a name, a collection of one or more (data) field declarations (i.e., variables) and method definitions. A field is a named holder of a value or object reference and can be used to represent state that persists over individual invocations of the methods of the class.

An **object** is an instance of a class that contains its own copies of the fields declared in its class and is able to execute the methods declared in its class on these fields. An object has a unique identifier which can be stored in variables and can be passed as method parameters (i.e., by reference). An object is said to exhibit the behaviour

declared in its class, using its own copy of the variables (as defined by its class) to store state changes over time.

A class may control how an instance (i.e., object) is created from it. In this case, classes specify one or more[1] special methods, also referred to as **constructors** or **initialisers**, which prepare an object for use, often by allowing the class's fields to be set to certain values via a list of formal parameters. Note that a constructor is used to create an instance, so formally as a method, it actually returns a single parameter – an instance of the class. Since constructors are essentially methods, an additional, important subtype of method invocations inside method bodies is a **constructor invocation**.

3.1.3 Interfaces and Types

An **interface** is a named collection of method signatures which defines the externally visible access points for invoking functionality defined by a class and delivered by objects. An interface itself does not define or imply any behaviour – it is purely structural. A class is said to be "compatible" with an interface if there are one or more possible bijective mappings between the method signatures in the interface to the methods in the class. In general, a class can have more than one interface since subsets of compatible method signatures of a class may be mapped to the method signatures in different interfaces.

A **type** serves to constrain the nature of the values or objects that can be assigned (or mapped) to parameters or variables. In general, there are two kinds of types (1) data types whose instances are immutable named values, and (2) object types whose instances are mutable objects. In the former, values are their own identifiers, while in the latter objects have unique individual identifiers which can be stored in variables and passed as parameters. There are two kinds of data types (1) primitive data types (e.g., predefined types such as integers etc.), and (2) compound data types (e.g., dates, strings etc.). Data types are characterised by the set of values they define, while object data types are defined by the functional abstractions they support [44].

3.1.4 Systems

A **system** is a behavioural actor that can respond to invocations. In other words, it can interact with one or more external actors (i.e., clients) in order to deliver services. Different kinds of systems can be used to realise actors, such as human beings and software systems. A **software system** is realised as a coherent collection of the

[1] Python officially only supports a single constructor, whereas Java allows multiple ones.

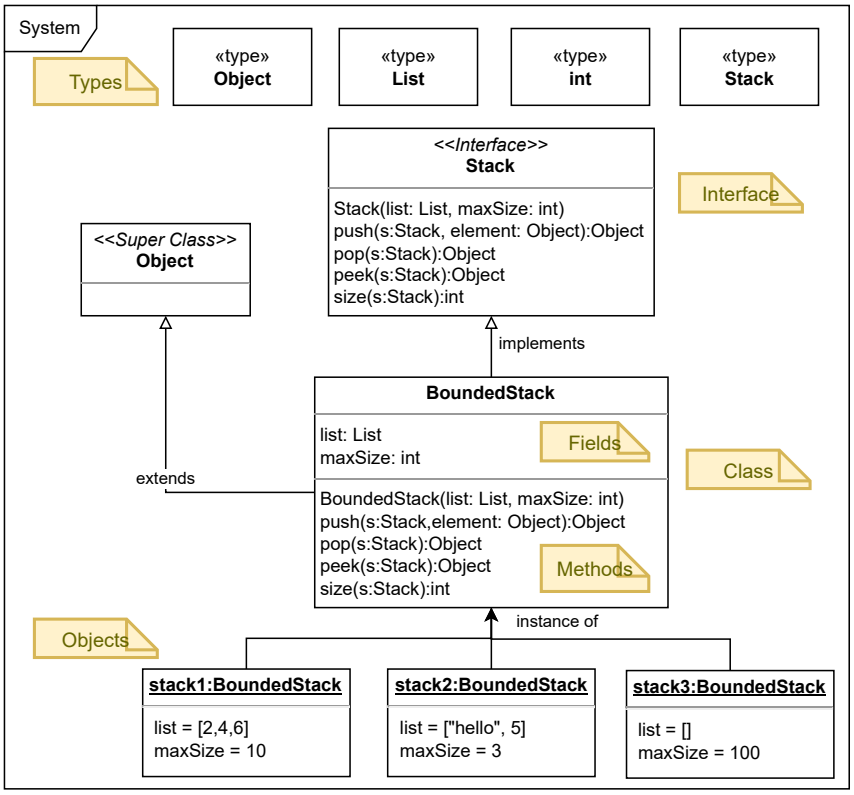


Fig. 3.1.: Basic Object-Oriented Notions - Stack Example

aforementioned object-oriented programming concepts and delivers its behaviour when executed on a computing platform and invoked through one or more interfaces. In other words, a software system has one or more interfaces by which external actors can access its delivered behaviour (i.e., by invoking the methods in these interfaces).

3.1.5 An Example - Stack

Figure 3.1 illustrates the aforementioned object-oriented concepts based on the example of (a variant of) the stack abstraction (see first motivational example in Section 2.1). It shows a software system which defines an interface to a stack functional abstraction, called `Stack`, as well as a class that implements that interface called `BoundedStack`, a bounded stack that limits the number of elements that can be pushed on a stack. The class declares two fields, a single constructor and four methods.

In addition to the class realisation, three objects are instantiated from the class (i.e., `stack1`, `stack2` and `stack3`). The class controls the way objects can be instantiated via its constructor called `BoundedStack`. In this case, it defines two

```
1  push(s:Stack, element:Object)->element:Object // signature
2  { // body (block of statements)
3      currentSize = size(s) // method invocation
4      if(currentSize >= maxSize) { // condition
5          return null // or other means of signalling
6      }
7      add(list, 0, element) // method invocation on (field) object 'list'
8      return element
9  }
```

List. 9: Stack Push Method Example (Pseudo Algorithm)

formal input parameters, `list` and `maxSize`, to initialise the stack using an existing list of elements as well as to define the capacity of a stack instance. Each object instantiated holds its individual actual values for the class's fields based on its current state after method invocations which may modify the values of the object's fields.

Since both fields and methods constrain their (parameter) values using types, this example uses four basic types: `Object`, `List` (ordered list of elements), `int` (integer value) as well as `Stack` (defined by the class). Note that the `Object` type is simply used as a placeholder to allow any value or instance of any type to be pushed onto the stack. `Object` is regarded as the super type of all possible types in the programming language[2].

Listing 9 illustrates how the stack's push method is implemented in a pseudo-algorithm like way. The method signature `push(s:Stack, o:Object)->Object` defines the method signature of the push method, whereas the statements inside the block, encapsulated by curly-braces, represent the operations invoked in the method body. Note that the subject of a method call is explicitly passed via the method signature as illustrated by the invocation of the `size` method in Line 3. Here the stack object (i.e., instance of class `Stack`) whose size is being determined is passed as the actual parameter, `s`, of the invocation of the method `size` which returns the current size of the stack.

3.2 Stimuli, Responses and Actuations

A system is actuated by means of **stimuli** which are sequences of one or more invocations of the methods in its interfaces. An individual **stimulus** is a sequence of one or more invocations of the methods in the interface(s) of a system, including the list of actual input parameters for each invocation. The smallest possible stimulus is thus the invocation of one method, with a particular set of input parameters.

[2] In Java, `java.lang.Object` is the super class, whereas in Python, `object`, is the super class.

In practice, inputs can take forms other than just the formal parameters. Global variables, class fields, files etc. can all provide contextual information that can influence the execution of a method. Without loss of generality, in the formal model we assume that these kinds of inputs are also modelled as explicit input parameters of methods (analogously to the way the state of an object is passed to a method as an explicit parameter in some programming languages and in our formal model). In other words, we assume any method can be written (or transformed) in such a way that all its "inputs" are represented as explicit input parameters.

A **response** is the set of outputs that occur when a system is stimulated through a stimulus (i.e., as a result of a stimulus being executed on a system). Note that in practice, as with inputs, outputs can take forms other than just the formal output parameters (e.g., fields, files etc.). However, as with inputs, without loss of generality, in the formal model we assume that these kinds of outputs can also be modelled as explicit output parameters of methods. In other words, we assume any method can be written (or transformed) in such a way that all its "outputs" are represented as explicit output parameters. Note that this includes the single "return" value supported in many object-programming languages (e.g., Java).

An **actuation** is a stimulus/response pair. In other words, an actuation encapsulates all the method invocations within a stimulus, including all the actual input parameter values, as well as the results and effects of the invocations, including the actual output parameter values. An actuation therefore contains a complete record of how a system was stimulated by a stimulus and how the system responded. An actuation is therefore essentially an **observation** of the system's behaviour in response to a particular stimulus.

Assuming, in the formal model, that all sources of influence on the behaviour of a system are represented as explicit inputs to methods in the interfaces of the system and that all results/effects of the behaviour of the system are represented as explicit outputs of the system, simplifies the controllability and observability of the system. This is because all states of the system (i.e., periods between method executions) can be captured by the full history of method invocations that give rise to it. In other words, the only states that can be seen from outside the system are those that can be generated by a particular method invocation sequence (i.e., stimulus) and observed by the outputs of the invocations in that sequence (i.e., the response). Systems are therefore controllable and observable at this level of granularity and only at this level of granularity (i.e., the set of methods defined in the interfaces and their explicit input/output parameters).

3.3 Sequences and Operations

A **method invocation sequence** is an ordered list of method invocations. It reflects the typical programming style used in classic object-oriented unit tests to formulate stimulations of the system under test (SUT). Formally, a method invocation sequence is an ordered list of method invocations $S =< m_1, ..., m_n >$ where m_i denotes a method invocation as defined above in the i^{th} position of sequence S.

The total ordering of method invocations in a sequence also defines the order in which they are executed on a software system s. We can use the indices of method invocations as coordinates to refer to a specific method invocation in the sequence and its recorded result (i.e., output) after execution (m_i where i is the i^{th} invocation of a method in the sequence). This is of interest in particular since often the outputs of some method invocations become the inputs for subsequent method invocations in the sequence (cf. basic principle in programming languages such as left-hand side variable assignments). Moreover, in testing, individual (intermediate) outputs may be used to obtain and observe the system's behaviour.

For the sake of formal completeness, and to cover other useful operations provided by object-oriented languages, we abstract the concept of an object-oriented operation to model useful lower-level expressions (i.e., statements) other than method invocations which are often found in test method bodies. An **operation** is an individual step (i.e., statement) in a method body that describes how the method's behaviour is realised. Similar to method signatures, formally, based on term algebra, an operation op has a type-signature $op : [I_1, ..., I_k] \rightarrow [O_1, ..., O_l]$, where $[I_1, ..., I_k]$ is the list of input parameter types, and $[O_1, ..., O_l]$ is the list of output parameter types. Each of these parameters can be accessed via its position through indices.

As a consequence, we can further generalise method invocation sequences into general-purpose "sequences" that cover other typical operations of object-oriented languages as well. Formally, a sequence (of operations) is an ordered list of operations $S =< op_1, op_2, ..., op_n >$ where op_i denotes an operation as defined above in the i^{th} position of sequence S.

For example, a method invocation of method `m` declared by class `A` which takes a single input parameter of primitive type `int` and returns an output of primitive type `boolean` (i.e., `A.m(int)->boolean`) is represented as an operation $m : [A, int] \rightarrow [boolean]$ where m denotes the name of the operation. For non-static (i.e., stateful) method invocations on a class instance (i.e., object), the first type of the operation is always the type of the method's declaring class (i.e., the receiver). The same applies to static method invocations as well, but the first type of the operation resembles the type of the class that declares the method, rather than the object.

A sequence of method invocations characteristic of the bounded stack example introduced above, for instance, can be formally represented in "term algebra" as follows (where the first invocation is a constructor invocation) —

$$
\begin{array}{llll}
m_1 & Stack: & [List, int] & \rightarrow [Stack] \\
m_2 & push: & [Stack, Object] & \rightarrow [Object] \\
m_3 & size: & [Stack] & \rightarrow [int] \\
m_4 & pop: & [Stack] & \rightarrow [Object] \\
m_5 & size: & [Stack] & \rightarrow [int]
\end{array}
$$

Note that state-of-the-art object-oriented languages such as Java and Python also realise the functional programming paradigm. Here, the operation model is applicable as well. Functions that are declared in the "global scope" (outside classes) are actually declared in the context of a "module". In this case, the type of the first input parameter is simply a "reference" to the "module" (package namespace) that declares the function (similar to static methods).

The concept of an operation is versatile, it can even represent the access of a constant value like an integer number $11 : [] \rightarrow [int]$ or the access of a field of a class $f : [A] \rightarrow [int]$ (i.e., access values of field `f` declared by class `A`).

3.4 Behaviour

The **behaviour** (a.k.a., semantics) of a system is formally viewed as the mapping between all possible stimuli that can be applied to the system and the responses of the system to those stimuli (as illustrated in Figure 3.2). In other words, the behaviour of a system is the set of all possible actuations of a system.

Formally, let mapping $A_s : S \rightarrow R$ define the behaviour of system s in terms of all its actuations (i.e., stimulus/response pairs) where S is the set of possible stimuli and R the set of possible responses of system s. An actuation (i.e., stimulus/response pair) is then defined as an ordered pair consisting of a stimulus and a response, $a = (x, A_s(x))$, where $x \in S$ and $A_s(x) \in R$.

In the case of software systems, these actuations are described in terms of sequences of invocations of methods in their interfaces. Every software system has one and only one "true" behaviour on a particular computing platform (i.e., execution environment), which is the behaviour it exhibits when executed on that platform.

Note that our formal definition of behaviour does not include non-functional properties of a system's execution on a particular platform (e.g., response speed, resource usage, reliability etc.), which may change from platform to platform. However, since all execution platforms for a particular programming language are assumed to correctly realise the semantics of that language, the (functional)

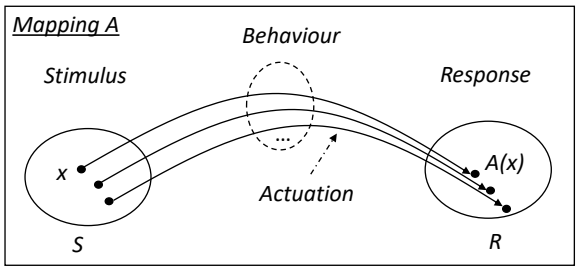


Fig. 3.2.: Relationships - Stimuli, Responses, Actuations and Behaviour

behaviour of a **deterministic** software system should be constant across all execution platforms for the language it is implemented in.

If all inputs to the system (i.e., all things that directly affect its behaviour when stimulated) and outputs of the system (i.e., all things that it affects when stimulated) are made explicit in the form of parameters, and the behaviour of a system is not constant across different executions, the system is said to be **non-deterministic** since there are no hidden variables that can account for differing responses to a given stimulus. In contrast, deterministic systems always give the same response for a given stimulus. For example, even a random number generator can be regarded as deterministic if the seed used to generate random numbers is defined as an explicit input parameter. However, this model of behaviour also allows non-deterministic behaviour to be modelled if one or more of the things that affect the behaviour of the system are not represented as explicit input parameters to the methods. Such variables are often called hidden variables in the literature. For example, if the seed used to generate "random" numbers by a random number generator is not made explicit, and in effect becomes a hidden input, the stimulus-response mapping is no longer unique over different executions and the method no longer deterministic.

In some (extreme) cases, a stimulus may cause a software system to execute indefinitely. Such a "behaviour" of systems may be either desired (e.g., infinite looping to listen for new connections) or undesired (e.g., infinite looping by accident). In order to simplify our model of behaviour, we assume that systems eventually halt and produce a response to a given stimulus. This assumption ensures that we can describe the complete set of actuations for systems.

3.4.1 Behavioural Equivalence, Subsumption and Similarity

For our purpose, there are three important relationships between behaviours —

1. *Functional Equivalence*: A behaviour, g, is regarded as being *functionally equivalent* to another behaviour, h, if they are equivalent – that is, if the set of actuations of g is the same as the set of actuations of h. Formally, let $A_g : S \rightarrow R$

be the mapping that describes the set of all actuations of behaviour g, and let $A_h : S \rightarrow R$ be the mapping that describes the set of all actuations of behaviour h. Then behaviour g and h are functionally equivalent if and only if $A_g(x) = A_h(x)$ for all $x \in S$.

2. *Functional Subsumption*: A behaviour, g, is regarded as being *subsumed* by another behaviour, h, if the set of actuations of g is a subset of the set of actuations of h. g is then said to be subbehaviour of h. Formally, let $A_h : S \rightarrow R$ be the mapping that describes the set of all actuations of behaviour h, and let $A_g : S' \rightarrow R$ be the mapping that describes the set of all actuations of behaviour g where $S' \subset S$. Then behaviour g is subsumed by behaviour h if and only if $A_h(x) = A_g(x)$ for all $x \in S'$. Note that a behaviour is subsumed by itself, since a set is a subset of itself.

3. *Functional Similarity*: A behaviour, g, is regarded as being *functionally similar* to another behaviour, h, if the intersection of the two sets of actuations of g and h is non-empty. Formally, let $A_g : S \rightarrow R$ be the mapping that describes the set of all actuations of behaviour g, and let $A_h : S \rightarrow R$ be the mapping that describes the set of all actuations of behaviour h. Then behaviour g is functionally similar to behaviour h if and only if $A_g(x) \cap A_h(x) \neq \emptyset$ for all $x \in S$. Accordingly, if the intersection of the two sets of actuations is empty (i.e., $A_g(x) \cap A_h(x) = \emptyset$), g and h are *functionally distinct* (i.e., since they do not share any subset of actuations). Note that $A_g(x) \cap A_h(x) \neq \emptyset$ defines another behaviour k that may neither be functionally equivalent to nor subsumed by behaviour g or h.

In general, it is not possible to establish (i.e., prove) any of these three relationships formally due to Rice's theorem [201].

The functional similarity relationship between two behaviours complements the other two relationships of functional equivalence and functional subsumption. Note that if g and h have the same set of actuations (i.e., $A_g(x) = A_h(x)$), g and h are functionally equivalent as well as subsume functional similarity (since $A_g(x) \cap A_h(x) = A_g(x) = A_h(x)$). If either g subsumes h, or g is subsumed by h, g and h are functionally similar (i.e., one set of actuations is a subset of the other).

3.5 Implementations, Specifications and Functional Abstractions

Although behaviours are formally defined as the set of all possible actuations of systems, only trivial behaviours can be fully described in this way in practice, because

the number of actuations (i.e., stimulus/response pairs) of non-trivial behaviours is too large to enumerate [6]. Therefore, in practice, behaviours also have to be defined by other means[3]. Approaches for defining behaviour can be divided into two groups —

- *Executable*: Executable descriptions of system behaviour are described in a language (e.g., a programming language) that is "understood" by a computing platform and can therefore be executed to create information and/or cause effects in the real world. An executable description of behaviour describes the "true" behaviour of the system realising that description, since the actuations that formally define that system's behaviour can, theoretically at least, be obtained by observing the executable description's (i.e., the system's) response to arbitrary stimuli. A concrete executable description of behaviour is therefore said to define the "true" behaviour of a system, which is equivalent to the actual behaviour (i.e., stimulus/response pairs) that the system exhibits when executed. Such a description is said to **implement**, or be an **implementation of**, that system and such a system is therefore said to be a **concrete** system. An executable definition of behaviour therefore implements (i.e., completely defines) the true behaviour of one, and only one, concrete system, which is the set of actuations obtained by executing it on a suitable computing platform.

- *Non-executable*: Non-executable descriptions of a behaviour describe the behaviour of an (abstract or virtual) system using a language which, although not executable, has sufficiently rich semantics to characterise the actuations (without enumerating them). These languages can be informal (e.g., natural language) or formal (e.g., logic, mathematics) (e.g., [112]), and can characterise the behaviour at different levels of precision. Non-executable behaviour descriptions are referred to as **specifications** of a behaviour. Common languages that can be used to create non-executable behaviour descriptions (i.e., specifications) include natural languages, mathematics, modelling languages and constraint languages etc.

3.5.1 Functional Abstraction

In contrast to an executable behaviour description (i.e., code) which defines the true (i.e., run-time) behaviour of a concrete system, a non-executable behaviour description describes the behaviour of an abstract or virtual system with no "true" behaviour per se. Rather than using the terms "abstract system" or "virtual system",

[3]Note that explicit enumerations of actuations are still useful in practical software engineering as partial descriptions of a system's behaviour.

however, we prefer the term **functional abstraction**. A functional abstraction, therefore, is a named behavioural abstraction (i.e., description of the behaviour of an abstract/virtual system), which defines the set of actuations executable on its interface in a non-executable way.

A simple example of a functional abstraction is the "sort" abstraction, which takes in an ordered list (or array) of comparable elements and returns an ordered list containing the same elements but in either ascending or descending order[4], and the "stack" abstraction introduced earlier which offers a set of methods for pushing elements onto, and popping elements off, a data structure in last-in-first-out order (LIFO).

Note that non-executable behaviour definitions (of functional abstractions) may be as, or less, precise than executable behaviour definitions (of concrete systems). By definition, a complete enumeration of all the actuations of a behaviour, when possible, provides a 100% precise definition of (the behaviour of) a functional abstraction. Given their formal semantics, behaviour definitions expressed using mathematical or logic-based languages can also provide 100% precise definitions of behaviour. However, behaviour definitions in natural language and/or informal notations such as UML [80], are usually less than 100% precise.

Since functional abstractions have an interface as well as behaviour, they can also be related by the functional equivalence and functional subsumption relationships defined above. However, stating that two functional abstractions are functionally equivalent to one another is effectively the same as defining another name (i.e., an alias) for the same abstraction. Thus, if the behaviour associated with functional abstractions "sort" and "order" are equivalent, "sort" and "order" are essentially aliases for the same functional abstraction. Subsumption is a more useful relationship between functional abstractions, and indicates that the behaviour of one is subsumed by the behaviour of another.

A simple example of a subsumption relationship of two functional abstractions is a method for adding together two numbers. One may specify a functional abstraction, "sum of positives", to only add together numbers which are positive (e.g., using a method signature `sum(a:int,b:int)->result:int`), while another functional abstraction, "sum of numbers" adds both positive and negative numbers. According to the definition in Section 3.4.1, the latter formally subsumes the former, since all the set of actuations of "sum of positives" is a proper subset of the actuations of "sum of numbers". The latter functional abstraction defines additional actuations related to summing up negative numbers.

[4]Oftentimes, either the natural ordering of the elements' type (e.g., numbers) is used or some custom ordering is defined.

An alternative subsumption relationship between two functional abstractions can be identified between a double-ended queue (also known as "deque") and a stack or a queue [147]. Since a deque effectively offers both stack and queue behavior, the deque abstraction subsumes both the stack abstraction and the queue abstraction. Accordingly, a deque defines method signatures for both a stack and a queue, but offers the capability to apply operations on both "ends" of the list of elements (i.e., LIFO for stack and first-in-first-out, FIFO, for queue).

In the rest of this thesis, the term "system" is regarded as being synonymous with "concrete system" unless otherwise qualified. Similarly, the term "behaviour of a system" is regarded as being synonymous with "true behaviour of a system" unless otherwise qualified.

3.5.2 Implements and Specifies Relationships

Although a concrete software system only **implements** (exactly) one behaviour directly (the true behaviour of the system) it can indirectly **implement** multiple behaviours associated with functional abstractions. More specifically, a concrete system, s, is said to implement a functional abstraction, f, if —

1. the method signatures defined in the interface of f, M_f, is a subset of the method signatures defined in the interfaces of s, M_s, and,

2. the behaviour of f, defined on the method signatures M_f, is subsumed by the behaviour of s, defined on the method signatures shared with f.

Formally, let $A_s : S \rightarrow R$ be the mapping that describes the set of all actuations of system s, and let $A_f : S' \rightarrow R$ be the mapping that describes the set of all actuations of functional abstraction f where $S' \subset S$. Then system s implements functional abstraction f if and only if $A_f(x) = A_s(x)$ for all $x \in S'$ over the (non-empty) set of shared method signatures $M_f \cap M_s \neq \emptyset$ where $M_f \subset M_s$.

In other words, if the behaviour associated with a functional abstraction is subsumed by the behaviour realised by a concrete software system, that system is said to implement, or be an implementation of, that functional abstraction. Note that formally, this includes functional equivalence. Based on the previous definitions, it follows that a functional abstraction can be implemented by zero or more concrete systems, and that a concrete system can implement one or more functional abstractions.

A functional abstraction f is said to **specify** a concrete system s if the behaviour of f, defined on the method signatures in the interface of f, is equivalent to the true behaviour of s, defined on the method signatures in the interfaces of s shared

with f. Thus, if f specifies s, then s implements f. However, the inverse is not the case. This is because the specifies relationship requires equivalence, whereas the implements relationship merely requires subsumption.

3.5.3 Oracles

In practice, the existence of implements and specifies relationships between systems and functional abstractions is a "belief" or "claim" rather than a demonstrable fact. This is a corollary of Rice's theorem which means that the equivalence of two behaviours is not, in general, computationally decidable, and is the source of many of the most challenging questions in software engineering, such as —

- does a system implementation match a specification?
- does a system implementation pass a set of tests?
- do a set of tests match a specification?
- does a specification match a set of tests?

In our model these questions are modelled by establishing subsumption and equivalence relationships between different kinds of behaviour descriptions associated with functional abstractions and systems, and the selection of one of them as representing the **oracle**, or trusted source of truth, which is deemed to be correct. Three forms of behavioural abstraction descriptions are typically used in practical software engineering projects —

(A) executable descriptions of concrete software systems,

(B) non-executable specifications of functional abstractions,

(C) explicit definitions of actuations of functional abstractions (typically called tests).

Because of the sheer number of actuations involved in all but the most trivial functional abstractions, descriptions of type (C) are usually incomplete.

In the context of the sorting functional abstraction mentioned previously, assume the goal is to sort a list of numbers in ascending order. An executable description of such a software system (A) that implements this functionality can be a single method with the method signature `sort(list:List)->result:List` that accepts a list of numbers and returns a sorted list. The method body may implement the sorting of a list of numbers using a dedicated sorting algorithm like merge sort (or any other sorting algorithm including insertion sort, bubble sort etc.). A non-executable

specification (B) of this sorting functionality can be written using natural language or specified via other modelling techniques, while an explicit definition of actuations of the functionality (C) can be defined as expected input/output mappings in a unit testing framework (e.g., method invocation `sort([3,2,1]` is expected to return `[1,2,3])`.

A classic software engineering project usually involves all three forms of behaviour descriptions at some point in the development process – a specification of the functional abstraction (B) that represents the desired functionality, an implementation of a (concrete) software system (A) that implements the desired functionality, and executable descriptions of actuations of the functional abstraction (C) that "test" the desired functionality. In terms of our formal model, the goal is to ensure that —

(i) the (true behaviour of) the implementation (i.e., concrete system) subsumes the (behaviour of) the specified functional abstraction,

(ii) the (behaviour of) the specified functional abstraction subsumes the (behaviour of) the functional abstraction defined by the tests, and

(iii) the (behaviour of) the implemented system subsumes the (behaviour of) the functional abstraction defined by the tests.

If the aforementioned subsumption relations do not hold, one of the involved behaviours must be declared to be the oracle (i.e., the trusted or desired source of truth). Usually, the specification is regarded as being the oracle (i.e., the trustworthy description of the desired functionality) with which the others must concur. However, in test-driven development [33], the tests are often regarded as the oracle, and in regression testing, realisations of previous versions of the system are regarded as the oracle. Note that there can be only one oracle at a given point of time.

Formally, an oracle o acts as a predicate of correctness over a concrete system or functional abstraction and its possible actuations. So $corr(s, f)$ is a theoretical predicate of oracle o to determine the correctness of the software system s over functional abstraction f based on a set of actuations. It is defined as follows: $corr(s, f) \Rightarrow \forall a_i \in A_o : corr(a_i, s, f)$ where A_o depicts all actuations defined by the oracle (possibly only a subset of all possible actuations of f, since oracles are typically incomplete). In other words, oracle o assigns the truth values true and false by comparing the actual observed actuations of s with the ones defined by the oracle (i.e., comparing the responses). If the predicate evaluates to true for all actuations defined, the oracle is believed to confirm that software system s implements functional abstraction f. Again, since the oracle is simply a behaviour definition, it may be derived from a specification of the functional abstraction, from a software system or from both.

Part II

Observation Arena

4 Sequence Sheets

A major obstacle in supporting the large-scale, dynamic observation of multiple software systems is the heterogeneous ecosystem of languages and tools used to define software stimuli (i.e., tests), record the results (i.e., responses) and extract useful information from them (see Problem *P4* in Section 1.2). Their realisations do not scale and are rarely compatible with one another.

In this chapter, we introduce sequence sheets and a special notation for representing them called "SSN" to meet this need and represent stimuli as well as actuations (i.e., stimulus/response pairs) of systems in the arena of the observatorium. The language is used to define test sequences and to record the exhibited behaviours (i.e., responses) of systems on which they are executed. In the following sections, we first introduce the interface notation used by sequence sheets, followed by the sequence sheet notation and an example. Thereafter, we explain the differences between two variants of sequence sheets, stimulus sheets and actuation sheets. Finally, we discuss the potential of sequence sheets with respect to their reuse potential.

4.1 Interface Notation

In this section, we introduce our notation for representing interfaces which consist of collections of method signatures. This is roughly based on the UML notation. Method signatures are used to specify the methods offered by functional abstractions as well as (concrete) software systems. The grammar of the interface notation is illustrated as a simplified BNF grammar in Listing 10 and is then discussed using examples. Note that the presented grammar is not strict, since it does not check for duplicate method signatures which are not allowed, of course.

As defined before, an **interface** in our notation is a named collection of method signatures. A functional abstraction is composed of a name, an interface and a behaviour defined on that interface. It is represented by its overall interface as follows —

```
Stack {
    push(s:Stack, element:Object)->element:Object
    ...
}
```

```
notation      : interfaceSpec? EOF;
interfaceSpec : NAME '{' methodSig* '}';
methodSig     : NAME ( '(' inputs? ')->' outputs? | '(' inputs? ')' );
inputs        : parameters;
outputs       : parameters;
parameters    : (simpletype | qualifiedtype | arraytype | namedparam) ( ',' (simpletype |
↪  qualifiedtype | arraytype | namedparam) )*;
qualifiedtype : NAME ( '.' NAME )*;
simpletype    : NAME;
arraytype     : (simpletype | qualifiedtype) ('[]')*;
namedparam    : NAME ':' (simpletype | qualifiedtype | arraytype);
NAME          : [a-zA-Z_] [a-zA-Z0-9_]*;
```

List. 10: Simple BNF Grammar for Interface Notation (ANTLR 4 Syntax [189])

The name of a functional abstraction may coincide with the name assigned to its interface (i.e., “Stack” or “Base64”). But since in general, names are assigned by humans individually, they may differ in practice (e.g., use of different aliases etc.).

A method invocation (e.g., through the push method signature of the stack abstraction) causes the method’s body to be executed in such a way that the actual parameter values in the invocation are mapped to the corresponding formal parameters in the method signature. The designation of the object that receives a method invocation is typically a design choice in programming languages and varies among languages[1]. As previously defined for method invocations (i.e., operations) based on term algebra, however, we do require the explicit passing of the receiving system.

Since we assume positional parameters where the order of parameters is well-defined and the number of parameters is fixed, we can also omit the names of the parameters. If a method signature has no return parameter(s), we can either explicitly use the type `void` like many programming languages do, or we can simply omit the return type for the sake of simplicity. Applying the aforementioned simplifications, we can simplify our notion of method signatures to the following —

```
push(Stack, Object)->Object
```

Note that since in our interface signature notation it is obvious that the method signatures defined for the “Stack” functional abstraction belong to it, we may sometimes simplify our notation by omitting the first parameter of the receiving system (which is described by the stack abstraction) —

```
Stack {
    push(Object)->Object
    ...
}
```

[1] In Java, the receiver instance is implicit (but can be referred to via keyword `this`), whereas Python explicitly models the receiver instance (usually called `self`).

A **class** can be regarded as a software system which realises a functional abstraction and has one or more methods[2]. Even though a software system may be realised by more than one class, it is usually possible to identify a single class which represents the core entry point to the behaviour that the software system implements (cf. Section 5.2). Accordingly, all the remaining classes are assumed to "interact" with the core class of the software system and interactions may be modelled via supplementary method invocations. This assumption builds on encapsulation [152], a core concept in object-oriented programming where a class "bundles" and hides an implementation, and also builds on common interpretations of units in unit testing (e.g., class unit that is tested).

Constructors

Since classes may define special methods, constructors, which are used to control the creation of an individual instance of a given class (i.e., an object), we use the name of the functional abstraction or class to support these special method signatures as part of the notion of functional abstractions and its interface —

```
Stack {
    Stack(List,int) // constructor
    push(Stack,Object)->Object
    ...
}
```

The interface of the stack abstraction now defines a special method signature called `Stack` which takes two formal parameters of type `List` and `int` (see bounded stack example in Section 3.1.5).

To be formally complete, a constructor invocation actually returns an instance of the type of its declaring class, so strictly speaking the following alternative notion achieves notational completeness: `Stack(List,int)->Stack`. In case there is no non-empty constructor defined (i.e., which takes no parameters), we can omit it in the interface specification (similar to implicit default constructors defined in the Java language).

Multiple Output Types

So far, we have demonstrated our notation based on a potential interface signature of a stack abstraction. The method signatures presented, however, do not define

[2]Note that, strictly speaking, an empty (sub)class has "default" behaviour which is inherited from the root object, so inherited methods count as well.

multiple output types. In order to demonstrate this case as well, let us consider the problem of finding the midpoint of two coordinates in the Cartesian coordinate system. For this functional abstraction, we have identified an interface named `CoordinateSystem` that specifies a single method signature called `midpoint`. It defines four formal input parameters that resemble the two, two-dimensional coordinates (`x1`, `y1`) and (`x2`, `y2`). The behaviour of the method is to return the midpoint, a coordinate that is reflected by two output types, `x` and `y`, as follows —

```
CoordinateSystem {
    midpoint(x1:int,y1:int,x2:int,y2:int)->x:int,y:int
}
```

This shows that the same notation is used for output parameters as for input parameters. Output parameters are separated by a comma as well. Optionally, we may decide to further simplify the method signature by removing the names of the parameters.

Object-Oriented Interfaces

At first sight, the interfaces of functional abstractions look similar to object-oriented interfaces, so why not use them? In fact, they can be treated as such and may be translated to them. The reason, however, is that they are unsuitable in our case, since they are typically incomplete, and hence not expressive enough. A Java interface, for example, may look as follows —

```
1  interface Stack {
2      Object push(Object o);
3      Object pop();
4  }
```

However, Java interfaces, in particular, are limited to the declaration of methods only, so they do not allow constructors to be declared such as `Stack(List,int)`. Python, as another example, does not offer the interface concept at all[3].

In an ideal world, we could simply "apply" the required interface to software systems which realise the functional abstraction. In practice, however, more work is required to realise interfaces in real programming languages, especially if we need to map (i.e., "adapt") the interface of the functional abstraction to the actual methods offered by a software system (see adaptation in Section 9.3).

[3]There are certain techniques to work around this limitation such as "abstract base classes" (abc).

4.2 Method Invocation Sequences

Sequence Sheet Notation (SSN) is a notation for describing method bodies which are composed exclusively of method invocations (including constructor invocations) in a way that can be represented in a spreadsheet style (i.e., tabular form). The notation was inspired by the test sheet approach of Atkinson et al. [14]. Given the assumptions made in our formal model of method invocation sequences (Section 3.3) and the fact that even variables of primitive types can be reified using wrapper classes (e.g., `java.lang.Integer` that resembles the `int` primitive), this notation allows any linear sequence of statements to be represented.

In this section we use the concept of method invocation sequences to model a set of stimulations and actuations of a set of software systems. In order to depict what is executed in a sequence sheet, we adopt the typical terminology used in practical unit testing of object-oriented software systems. The actual system tested (stimulated and observed) is generally referred to as the system under test (SUT).

4.2.1 Signatures of Sequence Sheets

Sequence sheets are methods which have both a signature and a body, and can be invoked like a normal method. The only difference is that their bodies are defined as rows in a spreadsheet using SSN. The signatures of sequence sheets are defined using our notation for method signatures in current object-oriented languages, but are enhanced to allow multiple outputs as well as multiple inputs to be represented in a uniform way. For instance, the following sequence sheet signature named `pushOneElement` defines one formal input parameter and is intended to stimulate a concrete stack class, `stack` —

```
pushOneElement(stack:Stack)
```

The type of the formal input parameter refers to the type of the functional abstraction of interest. We can invoke the method signature using a concrete implementation's class which shares the interface with the functional abstraction.

Formally, we define the signature of a sequence sheet as for normal methods. Accordingly, let $SS_o : [s, I_1, ..., I_n] \rightarrow [O_1, O_2, ..., O_m]$ be the signature of sequence sheet SS_o. It is "invoked" with actual parameters where s depicts the SUT and I_i the list of additional formal input parameters for that sheet.

4.2.2 Bodies of Sequence Sheets

A method body represented in SSN is simply referred to as a "sequence sheet" for short. As illustrated in Table 4.1, a method body is represented in a tabular form,

Tab. 4.1.: Structure of a Sequence Sheet Body

	O_1	...	O_i	M	I_1	...	I_i
1				m_1			
...				$m_{...}$			
n				m_n			

```
1  ssn              : methodinvocation+ BODYOUTPUT* EOF;
2  methodinvocation : output (', ' output)* methodname input (', ' input)*; // row
3  output           : actualparam; // of method invocation
4  methodname       : NAME | INITIALISER;
5  input            : actualparam; // to method invocation
6  actualparam      : COORDINATE | VALUE | BODYINPUT;
7
8  NAME             : [a-zA-Z_] [a-zA-Z0-9_]*;
9  INITIALISER      : 'create';
10 COORDINATE       : [A-Z]+ [0-9]+;
11 VALUE            : ...;
12 BODYINPUT        : '?' NAME; // passed to body
13 BODYOUTPUT       : '!' COORDINATE; // returned from body
```

List. 11: Abstract BNF Grammar for a Sequence Sheet Body in SSN (ANTLR 4 Syntax [189])

with typical spreadsheet-like labels (also referred to as coordinates) to identify the rows and columns. Listing 11 presents a simplified BNF grammar of SSN.

Each row represents a method invocation $m_i : [I_1, ..., I_k] \rightarrow [O_1, ..., O_l]$, where $[I_1, ..., I_k]$ is the list of input parameter types, and $[O_1, ..., O_l]$ is the list of output parameter types, with sequential flow of control starting at the top and moving downwards. The name of the method invoked in a row is given in a special column (i.e., M) that separates the input and output parameters which are given in other columns (to the right and left, respectively). The choice of assigning inputs to the right and outputs to the left of the method follows the typical structure of method invocations in state-of-the-art programming languages. One may realise them in opposite ways as done in the test sheet approach [14]. Input and output (i.e., return) parameters to and from the method body are represented using a special label (i.e., a question mark and exclamation mark respectively).

Since the external signatures of methods invoked in sequence sheets are the same as in other notations, the methods invoked in a method body represented using SSN can be implemented in any suitable language (e.g., Java), with the commensurate limitation on the number of output parameters[4]. This, in turn, means that methods represented in SSN can be nested (i.e., a method body represented in the SSN notation can call a method whose body, in turn, is written in SSN notation).

[4]The number of return parameters in Java is limited to 1 (assuming that `void` is an explicit output parameter type).

	A	B	C	D
1		create	?stack	
2		push	A1	“hello world”
3		peek	A1	
4		size	A1	
5		pop	A1	
6		size	A1	

Fig. 4.1.: Sequence (Stimulus) Sheet `pushOneElement(stack:Stack)`

Strictly speaking, to support observations, sequence sheets do not need a return type in general ($SS_o : [s, I_1, ..., I_n] \rightarrow []$), since they already hold method invocation records after their execution (these can be regarded as the sequence's outputs). However, to be formally consistent with our model, we allow sequence sheets to also mark certain method invocation outputs as return values (i.e., using an exclamation mark).

4.3 Example Sequence Sheet

Figure 4.1 depicts a simple sequence sheet for invoking an instance of the stack example. It has the method signature `pushOneElement(stack:Stack)` and depicts a typical “use” (i.e., stimulation) of a stack by invoking its methods.

Each row in the sheet resembles a method invocation to a stack which can be addressed by an integer (here starting at 1). The corresponding interface assumed for the given stack (i.e., functional abstraction) is as follows —

```
Stack {
    Stack()->Stack // empty constructor
    push(Stack, Object)->Object
    pop(Stack)->Object
    peek(Stack)->Object
    size(Stack)->int
}
```

The sequence (sheet) defines six method invocations. The first column contains the placeholders for the output parameters, the second column gives the name of the invoked method, and the other columns identify one or more actual input parameters. Since, in this example, no invocation has more than two inputs parameters, only two inputs columns exists to the right of the method column. The first of these identifies the object whose method is being invoked. Overall, the number of output (or input) columns is always determined by the method invocation(s) with the maximum

number of parameters. Here the maximum number of input parameters is 2 (i.e., the push method) and for outputs is 1 (all methods). That is why the sheet only has one column to the left of the method column `B`. Note that since we take advantage of the classic spreadsheet notation, we can leverage its column/row coordinates (e.g., `A1`) to reference values. Later, we will use a simple indexing schema of numbers.

The first method invocation in the sheet plays a special role since it creates or identifies the entity being invoked. The string `create` is a special keyword in SSN that denotes the instantiation of a class. In this case, we declare that an instance of the stack abstraction is required and needs to be returned. Note that, technically, instances of classes may be obtained by invoking compatible constructors or by other means (e.g., via the singleton or factory pattern, see Section 9.3.3).

The stack sequence sheet and its sequence of invocable method signatures can be formally represented in "term algebra" accordingly (see Section 3.3) —

$$
\begin{array}{llll}
m_1 & create: & [List] & \rightarrow [Stack] \\
m_2 & push: & [Stack, Object] & \rightarrow [Object] \\
m_3 & size: & [Stack] & \rightarrow [int] \\
m_4 & pop: & [Stack] & \rightarrow [Object] \\
m_5 & size: & [Stack] & \rightarrow [int]
\end{array}
$$

All the method invocations in the presented sequence sheet call a stateful `Stack` object that is referenced via its row and column coordinates (here it is stored in `A1`, since the first method invocation returns the corresponding object). In this case, only the `push` method is actually called with one additional input parameter, a constant value `"hello world"` of type `String`.

4.4 Stimulus Sheets and Actuation Sheets

The strength of the SSN notation is that it can be used to not only describe a stimulus of the object or objects under investigation, it can also present a record of an individual execution of the sheet. A **stimulus sheet** describes the execution steps that are taken when the sheet is invoked, but does not carry any information about a particular invocation. Only the actual values of the input parameters for each particular method invocation are expressed. The output values are not expressed, but may be still be referred to using the spreadsheet coordinates of cells designated as placeholders for output values. Stimulus sheets therefore correspond to the code describing what will happen when a method body is executed.

An **actuation sheet**, on the other hand, contains a record of the responses of the SUT in a particular execution of the method, as well as the stimulus information. Actuation sheets, therefore, augment the information in stimulus sheets with

	A	B	C	D
1	«instance»	create	?stack	
2		push	A1	"hello world"
3	"hello world"	peek	A1	
4	1	size	A1	
5	"hello world"	pop	A1	
6	0	size	A1	

Fig. 4.2.: Actuation Sheet `pushOneElement(pkg.ArrayStack)`

information specific to one particular invocation of the stimulus sheet. In other words, an actuation sheet is created from a stimulus sheet by executing that stimulus sheet on the SUT, and contains the output values as well as the input values. As defined in the formal model, therefore, an actuation includes both the input (i.e., stimulus information) and the output (i.e., response) information and thus defines the mapping between the two.

Figure 4.2 illustrates an actuation sheet generated from the stimulus sheet in Figure 4.1 by executing it on a stack implementation. In this example, the sequence sheet is instantiated on the sample Java class `pkg.ArrayStack`. For this, the input parameter placeholder `?stack` is replaced with the fully-qualified name of the target Java class. The response of each method invocation (i.e. output value) is stored in the first column *A*.

4.5 Parameterisation and Reuse

As explained earlier, sequence sheets are invoked through well-defined method signatures. A major advantage of signatures is that they allow for the parameterisation of sequence sheets

The set of stimuli of a functional abstraction invariably includes stimuli which involves the execution of the same sequence of methods from the interface of the functional abstraction, but with differing inputs. In common with the usual practice of representing commonly used algorithms as parameterised methods, a parameterised sequence sheet captures a sequence of method invocations that can be invoked with different input parameters.

A sequence sheet acts in the same way as a test method known from object-oriented unit testing. They support both the principles of “decomposition” and “abstraction” in object-orientation, but in this case to mainly realise stimulations. Stimulations can be split into reusable sequences. An object-oriented test is usually realised in terms of a method as well, having an invocable method signature and a method body which contains the method sequence (i.e., testing behaviour). Like a

	A	B	C	D
1		create	?stack	
2		push	A1	?element
3		peek	A1	
4		size	A1	
5		pop	A1	
6		size	A1	

Fig. 4.3.: Stimulus Sheet `pushOneElement(stack:Stack, element:String)`

method invocation sequence, a test method can also be parameterised to enable its reuse to represent sets of related stimuli.

Taking our previous stack example, we can make the sequences more abstract by making the value pushed to the stack a formal parameter of type `String` —

```
pushOneElement(stack:Stack, element:String)
```

Based on the concept of abstraction, many variations of the same sequence can be defined by simply passing a different actual parameter each time (as illustrated in Figure 4.3). In the example sequence, the value of the actual parameter `element` passed to the sequence can be accessed by using the question mark syntax (i.e., `?element`).

4.6 Slicing and Extending Sequences

Since the possible input space of a system is typically huge, so is its space of possible method invocation sequences. Similar to the subsumption relationships introduced for the behaviour of functional abstractions and systems, subsumption relationships for method invocation sequences exist as well.

These subsumption relationships can be linked to a system's state and state transitions. For many software systems, several "happens-before" and "happens-after" relationships can be determined. For example, before a (stateful) method of a class can be invoked, we have to create an instance of that class on which we can invoke that method. We need to invoke the constructor of the class concerned to return an instance of it (e.g., using the `new` keyword in Java).

From the basic characteristics of states and method invocations, it follows that any sequence containing that stateful method at hand needs an initialiser invocation first (i.e., `create` in SSN). Accordingly, many sequences may contain the same subsequence which is invoked in a consecutive order (e.g., bringing the system into some desired starting state or final state using prefix and postfix invocations).

Abstractly, a method invocation sequence describes a subset of the possible actuations of the system that can be used for one of four purposes. Based on Ammann and Offutt's classification of possible stimuli [6], there are four types of collections of method invocation sequences, each of which has a certain purpose —

- *Prefix Invocations*: Any collection of method invocations to reach a desired starting state in which the behaviour of the SUT can be observed,
- *Postfix Invocations*: Any collection of method invocations to reach a desired ending state of the SUT,
- *Behaviour Invocations*: Any collection of method invocations necessary to reveal its actual behaviour (i.e., actuations),
- *Exit Invocations*: Any collection of method invocations to terminate or "reset" the system to bring it into its initial state.

Unit testing frameworks such as JUNIT use special markers (i.e., annotations) to mark methods based on their intended purpose (e.g., `@Test` for verification sequences, `@Before` and `@After` for prefix- and post-fix sequences, and `@AfterClass` for exit sequences).

One way to cope with duplication in method invocation sequences is to use parameterised sequences (as defined above) in case the same sequence is used over and over again. In case there is a change in the method invocations in a sequence (i.e., execution scenario), we cannot reuse parameterisation.

This leads to the requirement to either **reduce** or **extend** existing sequences, to **slice** them based on certain criteria (i.e., potentially remove certain invocations), or to **combine** several (sub)sequences into a new sequence.

Formally, we can model subsumption among a pair of method invocation sequences as follows – $SS_j =< m_1, ..., m_l >$ is a subsequence of $SS_i =< m_1, ..., m_k >$ if the elements (i.e., method invocations) of SS_j have the same relative positions as the elements of SS_i, even though one or more elements have been deleted. Note that since a proper method invocation subsequence preserves the relative positions of its elements, it guarantees adjacent method invocations.

Depending on the position of a subsequence in a larger sequence, it is not guaranteed that the larger sequence exhibits the same subbehaviour. For example, if a subsequence sits in the middle of a larger sequence, the starting subsequence of such a sequence may lead to different intermediate system states, so the intermediate and final responses of the system may differ. In other words, a subsequence simply represents different actuations of the system than the larger sequence.

An example of a subsequence in our stack sequence sheet is that we only consider the first four rows. In that case, the sequence invokes only the `push` and `peek`

methods and then checks the size. The element on the stack is never removed, since the pop method is not part of the sequence. The subsequence, therefore, exhibits different behaviour for a stack implementation.

Subsequences or individual slices of method invocations can be reassembled in a variety of ways. Modern test generation tools, actually, capitalise on this idea in order to identify "good" test sequences automatically (e.g., RANDOOP [184] and EVOSUITE [84]).

5 Software System Boundaries

Software systems that participate in the arena of the observatorium are structured collections of code elements at various levels of granularity (i.e., classes, methods, statements, branches etc.). The engineers who created them are typically able to ascertain whether certain code elements participate in delivering certain behaviour of some functional abstraction (e.g., potentially facilitated by program comprehension techniques that support the cognitive process involved [247]). This ability establishes the basis for goal-oriented measurement of dynamic or static software metrics (e.g., to estimate software quality attributes of interest). If all the code elements are known, the definition of the "extent" (i.e., measurement scope) of the system is straightforward.

However, since the arena may potentially contain a large set of non-trivial, yet unknown software systems harvested from software repositories, relying (even partially) on human judgment to ascertain whether code elements participate in delivering the behaviour of some functional abstraction is highly inefficient and error-prone. The only practical approach to scale up the process of determining system boundaries to the needs of the observatorium, therefore, is to **automate** the process based on a well-defined measurement model that allows the specification of scoping criteria tailored to the analysis goals at hand (Problem *P3* in Section 1.2). Unfortunately, existing approaches lack a common terminology and a model to describe behaviour-aware scoping criteria for software systems. In this chapter, we therefore introduce common terminology and a measurement model in order to facilitate systematic and comparable behaviour-aware measurements.

5.1 Containment and Inclusion

The code-based realisation of an (object-oriented) software system includes elements at multiple levels of granularity (see object-oriented notions in Chapter 3.1). These elements are referred to as **software components**. At the larger level of granularity are components such as methods, classes and packages, and at the smallest level are components such as statements and variable declarations. The components in a system's realisation can be in **containment relationships**. For example, statements are contained within methods, methods are contained within classes, classes are contained within packages etc. The containment relationship is transitive.

The smallest element of a stimulus, as defined in Chapter 3, is a method invocation with a specific set of actual input parameter values. A software component within a system is said to be **involved** in the delivery of the behaviour induced by a method invocation, with specific input parameters, if there is a possible change that can be made to that component (including its deletion) that changes the delivered behaviour for the invocation in question. If a component is not involved in the delivery of the behaviour corresponding to a particular method invocation, it is said to be **superfluous** to that behaviour (cf. [140]).

The notions of involvement and superfluousness apply to stimuli as a whole and sets of stimuli. A software component is said to be involved in the delivery of the behaviour induced by a stimulus (i.e., sequence of invocations with particular input parameters) if there is a possible change that can be made to that component that changes the delivered behaviour, and superfluous to the realisation of that stimulus otherwise. Similarly, a software component is said to be involved in the delivery of the behaviour induced by a set of stimuli, if there is a possible change that can be made to that component that changes the delivered behaviour for that set, and superfluous to the realisation otherwise. Finally, a software component is said to be involved in the delivery of the behaviour of the system as a whole, if there is a possible change that can be made to that component that changes the delivered behaviour for all possible stimuli, and superfluous to the realisation of that behaviour otherwise. Software components that are superfluous to the system as a whole represent "dead code" which, if the system is only intended to support one functional abstraction, should probably be removed. In theory, they are harmless (e.g., may be "optimised away" by compilers [1]) but are nevertheless a source of possible faults (e.g., they add unnecessary complexity that decreases program comprehension or causes unintended behaviour in future code changes) [205].

The involvement relationship is a transitive relationship which is constrained by the containment relationship. More specifically, if a component c, is involved in the realisation of a behaviour (corresponding to a method invocation, stimulus or set of stimuli), components which are contained in c are also involved in the realisation of that behaviour.

5.1.1 Metrics

Metrics are properties that can be defined on functional abstractions and software systems. Metrics can be static in which case they are derived from the (static) description of the functional abstraction or systems, or they can be dynamic in which case they are derived from specific actuations of the software system. Dynamic

metrics can, therefore, only be the evaluated on software systems, not on functional abstractions. The calculation of metrics is constrained by different kinds of scopes.

Software Quality

A classic, but important field from which many relevant metrics for the observatorium can be systematically derived is the field of software quality. The need to estimate properties of software systems has been historically driven by the need to improve software quality (e.g., improve reliability and maintainability), and by the need to improve return on investments (e.g., efficiency) [37].

The area of software quality divides system quality into two basic types, *functional* quality and *non-functional* (or *structural*) quality. The former relates to what we refer to as the functional properties of a system (i.e., concerning its behaviour and the satisfaction of functional requirements), whereas the latter relates to what we refer to as non-functional properties of a system (i.e., the satisfaction of non-functional requirements). Software quality concerns are usually further split into distinct quality characteristics (also known as "ilities"), often represented as taxonomies or catalogues, that a software system has to meet (e.g., testability, understandability).

The work in the area of software quality spawned two core contributions: (1) software quality models that systematise software measurement (including international standards such as ISO/IEC 25010:2011 which replaced the standard ISO/IEC 9126), and (2) software metrics that have been proposed to quantify certain quality attributes of systems.

Over the years, numerous software quality models and metrics have been proposed to measure the "quality" of software systems [145], mostly in terms of non-functional properties measured by static metrics [227]. These include classic size-based metrics such as Lines of Codes (LOC), Halstead Complexity (HC) [105] and cyclomatic Complexity (CC) (e.g., [173]). For object-oriented systems, popular metrics include the well-known Chidamber and Kemerer (CK) metrics [50] and others [110].

5.1.2 Call Graphs

The invocation of a method causes the body of that method to be executed which, depending on its internal algorithm and the input parameters, may cause other methods to be invoked, in a nested fashion. If a method y is invoked by another method x, when x is invoked, y is said to be calledBy x within that invocation. The calledBy relationships between methods can be used to define a dynamic call graph that is based on a control-flow graph [5]. Since this graph is defined by the observed invocations that take place at run-time, we refer to it as an "*Observed Call*

Graph" (OCG). Its edges correspond to the calledBy relationships between method invocations originating from the method invocation in question.

Dynamic call graphs can also be defined at the level of stimuli by combining the OCGs of all the method invocations in individual stimuli. The OCG for a stimulus therefore contains all the nodes and edges of the invocations of the individual method invocations within it. The OCGs for individual stimuli can also be combined, in the same way, to create an OCG for a set of stimuli which contains the nodes and edges in the invocations of all the method invocations in all of the stimuli in that set. We represent such an OCG as $OCG(S)$, where S is the set of stimuli. The theoretical conclusion of this combination process is an OCG for all the stimuli in the actuation sheets defining the behaviour of the system, which we refer to as the "*Exhaustive Call Graph*" (ECG) for that system. Thus, the ECG for a system is defined by combining all the OCGs obtained from all possible stimuli of the system. However, for all but the must trivial systems, it is impossible in practice to make enough observations to create a system's ECG due to the vast number of stimuli defining their behaviour.

Due to Rice's theorem it is also impossible to derive a system's ECG analytically. However, it is possible to analytically estimate a system's ECG based on the syntax of its source code elements, although such algorithms are inevitably imprecise, because they cannot detect irrelevant or dead code reliably (which leads to an over-approximation of the ECG), and they may miss "dynamic" method invocations [38, 160] (which leads to an under-approximation of the ECG). We refer to call graphs determined in this manner as "*Syntactic Call Graphs*" (SCGs). Note that SCGs are static call graphs because they are created by means of static analysis of the code elements, while OCGs are dynamic call graphs, because they are constructed from observations of the execution of the system. Strictly speaking, ECGs are neither static nor dynamic because they are theoretical constructs which cannot, in general, be created by either dynamic or static means.

5.2 Scopes

The calculation of metrics is governed by the notion of **scopes**. Scopes are defined on the software components involved in realising some behaviour of interest that satisfy certain criteria. The criteria can be based on numerous things including depth in a call graph, ownership, source (i.e., origin) etc.

In the following, we assume that scoping criteria are applied to OCGs that are generated by observing the system's execution through a set of stimuli S. We use the notation, $scope_S$, in order to refer to scoping criteria that are applied to the call graph $OCG(S)$. If the set of stimuli corresponds to the set of all possible stimuli of

the behaviour at hand, then graph $OCG(S)$ depicts, theoretically, the ECG of that behaviour.

Depth-Based Scope

The behaviour of a system s is usually accessed by invoking the entry methods based on the functional abstraction the system is designed to realise with a set of stimuli S. Each method call in the call graph can be assigned a depth in terms of its shortest distance from the initial entry method call (i.e., the number of calledBy edges that have to be traversed). Using this notion of depth, depth-based scopes for defining metrics can be defined as follows —

$scope_S(d)$, where d is an integer representing a depth, is defined as including all software components within the bodies of all methods that have a depth d relative to the entry methods of system s, where the entry methods themselves have a depth of 0.

Thus, $scope_S(0)$ would calculate metrics only on the software components in the bodies of the entry methods involved in the realisation. On the other hand, $scope_S(1)$ would calculate metrics on the software components in the bodies of the entry methods, and in the bodies of methods that are called by those, but no others, and so on. Ultimately, $scope_S(*)$ calculates metrics on all methods in the call graph of the system's entry methods, that are involved in the realisation of the behaviour of system s. While it is possible to calculate metrics on different scopes for the different entry methods realising a system, usually the same scope is used for all of them.

Containment-Based Scope

Containment-based scopes are similar to depth-based scopes, but the criteria for inclusion in the calculation of a metric is based on containment within a larger component, which in the case of methods is classes or packages.

Thus, the **class scope** on a system, s (denoted $scope_S(class)$), would calculate metrics on the software components in the bodies of all methods involved in the realisation of s, that belong to the classes to which s belongs and are in the call graph of s. Similarly, the **package scope** on a system, s (denoted $scope_S(package)$), would calculate metrics on the software components in the bodies of all methods involved in the realisation of s, that are within the packages containing s, and are in the call graph of s.

Source-Based Scope

Source-based scope is similar to containment based scope but uses the slightly different notion of where a software component actually comes from (i.e., origin). The source of a component is intended to be the development context or delivery module it came from, which in the Java world are variously referred to as development kits, (third-party) libraries, applications, projects or frameworks. In terms of Java constructs, they take the form of one or more packages that are often combined, for example, into a downloadable "jar" file.

Suppose for example, that a system is implemented by several modules (i.e., groups of packages) – an application, a, developed in-house for the project in question, libraries, x, y and z, which are called directly or indirectly by a, and utilities package, u, which is the standard Java utilities package (i.e., from the Java Development Kit, in short JDK). Based on these different sources for the software components involved in delivering the behaviour of the systems, example scopes include —

- $scope_S(a, x, y, z, u) = scope_S(all)$: when applied to a system, s, this calculates metrics on the software components in the bodies of all methods involved in the realisation of s, that are in the call graph of s, from within all modules (i.e., the whole system).
- $scope_S(a, x, y, z)$: when applied to a system, s, this calculates metrics on the software components in the bodies of all methods involved in the realisation of s, that are in the call graph of s, from within the modules a, x, y and z but not u (i.e., components from the JDK are excluded).
- $scope_S(a, x)$: when applied to a system, s, this calculates metrics on the software components in the bodies of all methods involved in the realisation of s, that are in the call graph of s, from within the modules a, x but not z and u (i.e., components from the JDK, and a third-party library such as a common logging framework, are excluded).
- $scope_S(a)$: when applied to a system, s, this calculates metrics on the software components in the bodies of all methods involved in the realisation of s, that are in the call graph of s, from within the modules a only (i.e., only considers components from the project of the system).

Arbitrary Scopes

In general, there is no limit to the different kinds of criteria that can be used to define scopes that constrain the set of software components of a particular type used

to calculate metrics. For example, one could select all the statements that access a particular variable. This quickly overlaps with, or gets into, the area of program slicing where custom slicing criteria is used to determine all software components (cf. [250, 254]).

Finally, it is important to note that many current analysis tools and methods only consider limited kinds of scopes. Classic complexity metrics like McCabe's cyclomatic complexity [173] are often only calculated on the entry methods of a system (i.e., depth $scope_S(0)$), or on the methods in a class (i.e., $scope_S(class)$). We refer to these scopes as "shallow" scopes, since they only take into account a call depth of 0.

5.3 Measurement Approach

Having introduced our terminology and model for identifying the extent of the software within a system that should be analysed in a particular scenario, we now explain the principles of our measurement approach in greater detail. In order to integrate behaviour-aware scopes in the observatorium, three key requirements need to be met —

- *Definition/Selection*: Goal-oriented measurements of dynamic as well as static properties need to be supported which require the application of selected metrics of interest within different scopes,
- *Measurement*: A scope-aware measurement process for selected metrics needs to be established that scales to the needs of the arena, including the efficient collection and storage of obtained measurements,
- *Analysis and Interpretation*: An efficient post-processing and aggregation pipeline is required to enable the data-driven analysis of measurements.

To support a large number of potential software metrics, the measurement approach must be extensible in a way that allows (virtually) any software metric of interest to be integrated. This requires a flexible scope model that allows the integration and measurement of new metrics. In theory, the measurement of software metrics at a large scale is no different to the measurement at the scale of single systems. However, measuring scope-aware metrics on large sets of systems is a non-trivial endeavour, since it is necessary to deal with the idiosyncrasies of systems mined from large repositories (see Section 7.2). In order to be scalable, the logistics involved in the collection of measurements must therefore be highly efficient. Once the approach is able to collect large numbers of measurements, there must be efficient ways to post-process and aggregate then to support the decision-making processes of observatorium users.

In the following, we discuss the concepts behind the measurement approach supported by the observatorium that satisfies the aforementioned requirements. First, we introduce the paradigm that underpins our process of defining scopes for selected metrics of interest. Then we explain how the measurement and collection of scope-aware metrics are realised from an abstract point of view.

5.3.1 Process and Measurement Model

The conceptual paradigm that underpins our process for making scope-aware measurements is the goal/question/metric paradigm (GQM) introduced by Basili in 1994 [28]. GQM was developed to support a goal-oriented approach to software metrics. Its aim is to measure and improve the quality of software systems.

We propose GQM as a process and measurement model to systematically define the goal(s) of the analyses and measurements conducted in the observatorium. GQM defines three measurement models, each at a different level, that are tailored to the needs of the arena —

1. *Goal* (Conceptual Level): Definition of measurement goals when analysing software systems in the arena (depending on the task at hand),
2. *Question* (Operational Level): Characterisation of the code elements and their granularity (i.e., scope) in the context of certain dynamic (or static) properties from the viewpoint of the software systems,
3. *Metric* (Quantitative Level): Selection of suitable metrics to provide quantitative answers to these questions.

An alternative interpretation of the GQM paradigm is that it offers a systematic divide-and-conquer strategy to facilitate individual analyses in the arena. Users of the observatorium first need to define the goal(s) of their analysis. A *measurement goal* from the domain of software reuse, for instance, is to determine the "best software system" realising a certain functional abstraction (e.g., Base64 encoding). Intuitively, we may ask (the *question*) how "best" is determined for software systems. Assume the simple case in which "best" is determined through the classic property of complexity (i.e., system size) that typically affects the human understandability of software systems. Apart from the property of interest, we have to define the scope of code elements (i.e., software components) that are included in the measurement process. As explained previously, scope definitions are tailored to the underlying analysis. A particular user, may decide to limit his/her measurements to the project scope only (i.e., excluding any code elements that originate from third-party libraries). A simple *metric* to provide a quantitative measure of system complexity is LOC. The

measurements obtained eventually lead to a quantitative indication of the complexity of the systems under analysis.

5.3.2 Scope-Aware Measurements

In the following, we explain from a high-level point of view how we realise scope-aware measurements for software systems. First, we explain how we determine the boundaries of systems in terms of the software components (i.e., code elements) *involved* in the delivery of the behaviour of interest, and which are *superfluous* to that behaviour. Second, we explain how the scoping criteria are applied to the software components in order to select those of interest. Finally, we discuss how the software metrics can be applied to software components accepted by the scope criteria.

Figure 5.1 illustrates how scope-aware measurements are supported in the observatorium. To simply our example, we assume that the systems are functionally equivalent to the behaviour of a certain functional abstraction at hand, and that the set of entry methods of each system matches the interface of the functional abstraction.

Input: Entry Methods E of System s, Scope Criteria α, Metric β
Output: Measure m
1 **Function** `Measure(`E, α, β`)` $\rightarrow m$**:**
2 $M \leftarrow \emptyset$
3 **for** $e \in E$ **do**
4 $M \cup CalledBy(e)$
5 **end**
6 $C \leftarrow \emptyset$
7 **for** $m \in M$ **do**
8 $C \cup Elements(m)$
9 **end**
10 $C_\alpha \leftarrow Scope(C, \alpha)$;
11 $m \leftarrow Apply(\beta, C_\alpha)$;
12 **return** m

Fig. 5.1.: Pseudo Algorithm for Scope-Aware Measurements

Formally, let E be the set of entry methods of a system s that has been matched to the interface of functional abstraction f and that delivers the behaviour of f. The system boundary of s is determined by the set of all (transitive) methods, M, that are "involved" in the delivery of the desired behaviour. M is resolved through SCG analysis of "calledBy" relationships for each entry method $e \in E$ of system s.

For each method $m \in M$, we obtain a super set C of all software components (i.e., code elements) in their method bodies. Note that since C was determined by

SCG analysis at the method call level, it may still contain code elements (e.g., in the method body) that may not be "involved" in the behaviour of interest (i.e., are "superfluous" since they are not executed). Picking the method level of granularity in order to determine C is a "natural" design decision we made based on the typical encoding of functionality at the method level. As a consequence, the decision to accept/reject certain (un)executed code elements below the method level of granularity (e.g., statements) is postponed to the scope filtering step. In other words, it is possible to define precise scoping criteria based on the "involved" relationship that filters out superfluous code elements below the method level. Effectively, this becomes a low-level dynamic call graph analysis where the (non-)executed code elements are determined.

The next step is to filter out the code elements in C based on a set of scope criteria α defined by a user. The resulting set C_α is the filtered set which only contains all those code elements that were accepted by α. Finally, metric β can be computed based on the filtered software components C_α.

5.4 Quality

Behaviour-aware measurement scopes are widely applicable, since they allow a large range of focused analyses. On the one hand, assumptions about the measurement process can be "streamlined" and reused in and across analyses. On the other hand, the flexibility by which new measurement scopes can be defined, enables custom analyses to be conducted that require custom scopes. Similarly, the scope definitions allow for both the creation of "de-facto" scopes required by existing state-of-the-art tools and techniques, but also allows users to efficiently integrate their own metrics into a well-defined measurement process.

5.4.1 Sensitivity

It is important to note that measurement scopes have a huge impact on the measurable indicators of quality attributes, in particular of indicated software and test set quality.

Using custom scopes not only affects the comparability of results (i.e., measures may differ across a set of different scoping criteria) such as the study results obtained in software experimentation, but it also complicates the interpretation of metric indicators of software quality. In the long run, given the absence of standardised, de-facto definitions of system boundaries, however, it is desirable to establish a (pre-)defined list of scopes (e.g., based on popularity inferred from practical projects and/or studies).

Behaviour-aware system boundaries are also sensitive to the test sequences (i.e., stimuli) that were used to determine them. The risk here is that because of the limitations of dynamic analysis approaches (Section 16.1), software components of a system that are deemed to be relevant to the behaviour of the functional abstraction at hand may be missed. By implication, test sequences (e.g., sequence sheets) of a certain functional abstraction need to be carefully selected to better approximate the boundaries of matching software components in selected software systems.

5.4.2 Soundness and Precision

As explained before, scope-aware measurements require the "hybrid" combination of sophisticated SCG- and OCG-based analyses. Once the set of entry methods of all systems in the arena have been determined, SCG analysis can be applied in an "offline" manner. This does not require the actual execution of the system, it only requires knowledge of the set of entry methods and the project context of each system (i.e., all classes and libraries available for it) in order to compute the call graph.

Static analyses, however, can become costly depending on the chosen level of granularity of code elements (e.g., class, method, line or instruction level). The chosen level of method calls in our approach denotes a suitable trade-off between the soundness and precision (cf. Section 16.1) of the analysis with respect to our main objective, while at the same time keeping the cost of the analyses manageable. This makes it efficient enough to scale to many systems. While in our case static graph analysis typically over-approximates the boundary of a system with respect to the delivery of some desired behaviour (i.e., in addition to "involved" code elements there could still be "superfluous" code elements inside method bodies), it increases the soundness of our approach.

Precision in our scope-aware measurement is further increased in the second call graph analysis step. The dynamic call graph construction at the fine-grained level of code elements (e.g., down to the byte code instruction level in Java, for instance) keeps track of all executed code elements. In contrast to SCG analysis, OCG analysis depends on the execution of systems and is primarily driven by code instrumentation. Code instrumentation, however, does not come for free. It typically adds a significant overhead to the execution of software systems, since it consumes additional computing resources to produce potentially large execution traces that need non-trivial post-processing [255].

Note that because of the limitations of Rice's theorem, and the limitations of static and dynamic analysis approaches in general (sound, but imprecise vs unsound but precise), we attempt to achieve a synergy between the two by combining them

efficiently. Moreover, our measurement model is flexible enough to allow users to optimise and fine-tune the observatorium for either soundness or precision at the expense of increasing or decreasing the cost induced by scope-aware measurements.

6 Stimulus Response Matrices

In order to realise the envisaged observatorium, we need a unified approach that employs a dedicated data structure and associated conceptual model across the whole stimulus/response process (see Problem *P4* and *P5* in Section 1.2). In this chapter we introduce the notion of stimulus matrices (SMs) and stimulus/response matrices (SRMs) to meet this need. These describe the input to, and output from, the "arena" in which the large-scale observation of software systems takes place. As their name implies, these are matrices composed of stimuli and stimulus/response pairs (i.e., actuations, see Chapter 3). In the observatorium, these are defined in the form of sequence sheets using SSN. Together, SRMs and sequence sheets offer a navigational model to set up configurations of systems and tests, and to systematically store the execution results. We explain how the SRMs and sequence sheet data structures are leveraged to support large-scale data analytics for software systems, including their exhibited behaviour and execution metrics (Chapter 5).

6.1 Stimulus Matrices

As their name implies, stimulus matrices (SMs) are two-dimensional collections of stimuli represented in row and columns. While there is no rule as to the nature of the individual stimuli in a stimulus matrix and how they are organised, typically all the stimuli in a row are executions of the same sequence sheet, with the same actual input parameters except the parameter identifying the software system. Since the same sequence sheet is executed on different systems, this implies that all systems have to implement the same interface that is expected by that sequence sheet. In many scenarios, the interface is specified based on a functional abstraction of interest. On the other hand all the stimuli in a column are invocations of different sequence sheets, or the same sequence sheets with different input parameters, except that the parameter identifying the system is the same. Thus, all the cells in a row contain the same stimuli, but just applied to a different system.

A stimulus matrix can be viewed at two levels of abstraction – a *black box level* where just the method invocation is shown with the respective actual input parameters as illustrated in Figure 6.1, and a *white box level* where the full method invocation sequence (i.e., stimulus) is shown with all the inputs and outputs of all the contained method invocations as illustrated in Figure 6.2.

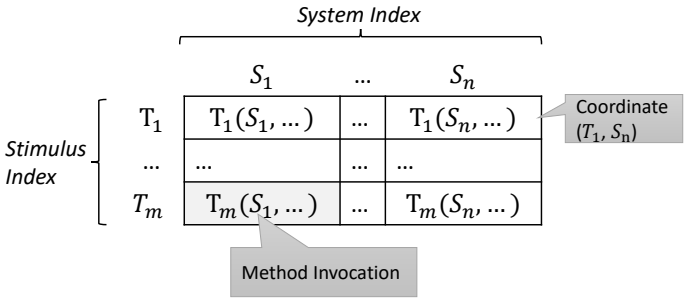


Fig. 6.1.: Stimulus Matrix - Black Box View (Sequence Sheet Invocations)

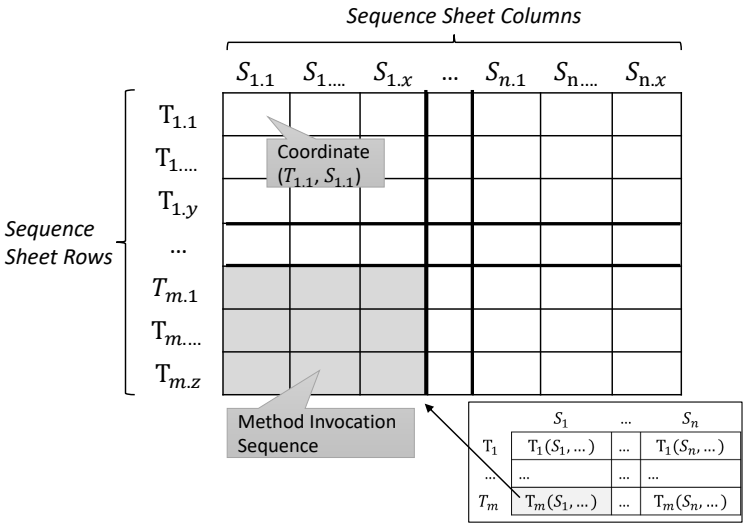


Fig. 6.2.: Stimulus Matrix - White Box View (Expanded to Method Invocations of Sequence Sheets)

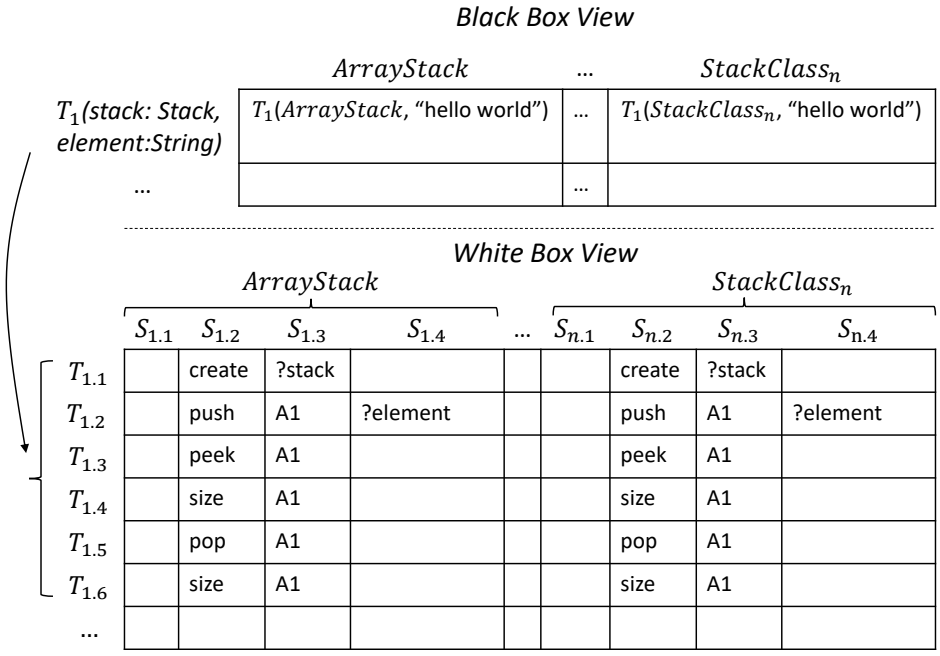


Fig. 6.3.: Stimulus Matrix - Black- and White Box View Example

The rows and columns of a stimulus matrix have well-defined numerical indices (i.e., coordinates) in order to access their contents. While in the black box level view, an element (T_i, S_j) of the stimulus matrix actually represents the invocation of a sequence sheet T_i on a system S_j, in the white box view an element $(T_i.o, S_j.p)$ in the matrix represents an element (o, p) of sequence sheet T_i on a system S_j (including the inputs and outputs, see Section 4.2). A sequence sheet row $T_i.k$ on columns $S_j.1$ to $S_j.l$, therefore, represents the invocation of a particular method at the k^{th} position of sequence sheet T_i on system S_j.

Since a stimulus matrix can contain different sequence sheets of varying length (method invocations and inputs and outputs), the white box view matrix may be sparse (i.e., contains empty elements). A black box stimulus matrix, on the other hand, is dense, since it only represents the invocation of sequence sheets on systems (i.e., the method invocation signature).

6.1.1 Stack Example

Stimulus matrices specify the configuration of invocations in the arena to describe what systems have to be stimulated and how they are stimulated. The indexing of stimulus matrices as well as sequence sheets provides fine-grained access to stimulus records.

Figure 6.3 illustrates two stimulus matrices that configure stimulations (i.e., invocations) for our stack example. There is one sequence sheet T_1 and several systems $S_i \in S$ defined (two of which are elaborated for demonstration purposes). The first stimulus matrix provides a black box on the number of corresponding invocations (i.e., combinations of $sequence\ sheets \times systems$), whereas the second stimulus matrix expands the view down to the method invocation level of sequences (i.e., white box view).

As we can see, both the sequence sheet invocations (black box) and the method sequence invocations (white box) are "instantiated", so all formal parameters have been replaced by their actual parameters based on the system of each column.

6.2 Stimulus Response Matrices

As their name implies, stimulus response matrices (SRMs) (a.k.a. actuation matrices) are two-dimensional collections of actuations represented in rows and columns. The only difference between SRMs and SMs is that the latter only contain stimuli (i.e., the stimulus information) while the former contain the full actuation information in terms of sequence invocation records and method invocation records in particular.

As with SMs, SRMs can be viewed at two levels of abstraction – a black box level where just the sequence invocation is shown with the respective input and output parameters, and a white box level where the full method invocation sequence record is shown (i.e., actuation record) with all the inputs and outputs of all the contained method invocations.

As well as the amount of information that is shown, the key difference between SMs and SRMs is that the former can be defined before any systems have been executed in the arena, while the latter can only be created by executing the systems in the arena with the stimuli in an SM. An SM therefore essentially represents the input to a run of the arena, while an SRM represents the output (i.e., the result of executing an SM). Moreover, the information in the input SM is a subset of the information in the output SRM. Formally, a stimulus matrix M_{SM} is transformed into a stimulus response matrix M_{SRM} via a function $f : M_{SM} \rightarrow M_{SRM}$ which maps invocation/system pairs to records.

The arena, however, is not the only agent that can add information to an SRM. If it is desired to have information from a human oracle in the actuation records for the functional abstraction of interest, as well as actuations arising from the execution of the implementations, the human oracles can directly "fill out" (i.e., add) the expected response information themselves. In general, therefore, the response information in an SRM can be added by the arena and human agents.

6.3 Analysis Attributes

As explained above, an SRM captures the results observed by executing the corresponding SM in the arena. A basic requirement of the observatorium is therefore to store records so that they can later be analysed. However, as we learned from the discussion of system boundaries and measurement in Chapter 5, there are observations other than actuations that are related to sequence or method invocations such as "non-functional" software metrics. To store observations of any kind, the SRM data structure allows additional custom "analysis" attributes to be stored. In other words, actuation sheets as well as method invocations are represented as "nodes" that can store additional attributes for later analysis purposes.

An (observational) attribute is simply a key-value pair. The key refers to the identifiable type of the attribute whereas the value may represent any arbitrarily complex value. There are many possible reasons to add such a pair. For example, the value could be a reference to an execution trace file, or it could be a constant value such as a string value or a numerical value representing a (scope-aware) measurement (Chapter 5). By having such a general model, SRMs can integrate any number of attributes both at the actuation sheet level and at the finer-grained method invocation level. This helps to establish a common "sink" to store any execution traces observed at execution time as well as the relationships between them. Note that apart from dynamic data, SRMs can also store any other data of interest including statically measured properties, or data computed by users as part of LSL pipelines (Chapter 10).

Note that in contrast to SRMs, SMs only store stimulation-related data. Since SMs and SRMs are technically closely related, we also allow the storing of attributes in SMs. These attributes are then simply copied over to the corresponding SRM after execution (if needed). This allows important metadata of interest (e.g., specific to the analysis and invocations at hand) to be made available after execution in the arena in order to improve efficiency in the post-processing of SRMs. The navigational model and the schema introduced in subsequent sections apply to SMs as well. But for the sake of simplicity, we refer to the more general notion of SRMs. Customised analysis attributes can also be stored by SRMs.

6.4 Navigational Model

The basic idea behind using record nodes and node attributes is to provide an intuitive model for navigating around SRMs and their sequences. For this, we represent the entities of an SRM as a hierarchical tree structure of connected nodes as illustrated in Figure 6.4.

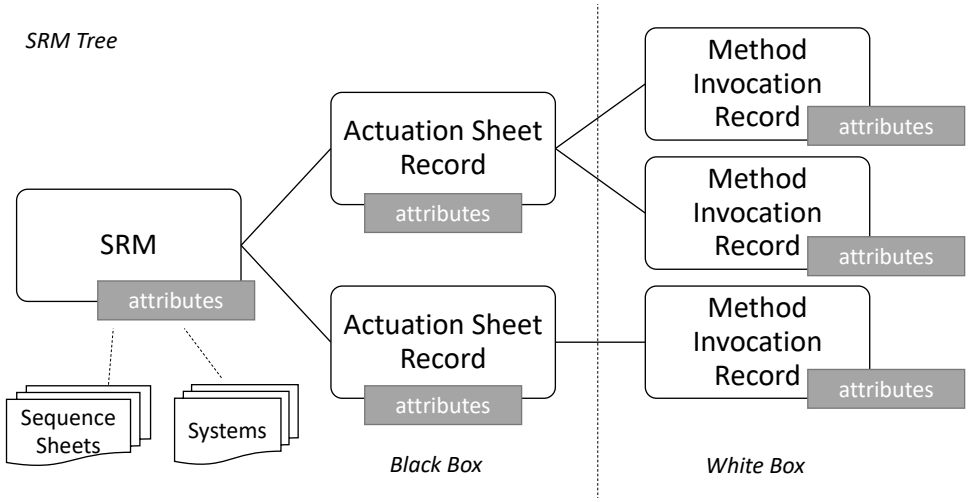


Fig. 6.4.: SRM Tree for Path Notation

An SRM represents the root node of the SRM tree. Its direct successors are the actuation sheet records which in turn have method invocation records as their children. The tree incorporates both the black box and the white box view depending on the level viewed. The black box view simply ends at the second level of the tree, whereas the white box level includes the third level.

Apart from "core" attributes such as a name and an identifier, an SRM tree node also holds a list of analysis attributes as discussed previously. Based on the simple SRM tree representation, a path notation including a set of path expressions can be defined in order to query, extract, transform and analyse data in an SRM. It is important to stress that there are two main reasons why a systematic navigational model is required in order to analyse SRM and actuation sheets efficiently —

- *Analysis Pipelines (Online)*: SRM navigation to support "online" analyses as part of analysis pipeline executions (Chapter 10).
- *Data Analytics (Offline)*: data-driven SRM navigation to support "offline" data-driven analyses in data analytics tools (e.g., R or Python's PANDAS [186] based on data frame manipulation) (Chapter 11).

For the former, we developed and integrated a path notation called SRMPATH into LASSO's LSL pipeline language, whereas for the latter we rely on tabular representations together with OLAP techniques such as pivoting (e.g., cube) to transform and analyse data.

The proposed SRMPATH notation serves as the formal basis to build more powerful path expressions that we leverage in LSL pipelines. In general, the most-widely used path notations are the dot notation (as known from object-oriented programming

```
// navigate records in tree top-down to leaf nodes
srm[srm_id].sequences[[sequence_id,system_id]].methods[row_id]

// core attributes ('attr' depicts the name of the attribute)
srm[srm_id].attr
srm[srm_id].sequences[[sequence_id,system_id]].attr
srm[srm_id].sequences[[sequence_id,system_id]].methods[row_id].attr

// analysis attributes ('attr_id' depicts the identifier of the attribute)
srm[srm_id].attributes[attr_id]
srm[srm_id].sequences[[sequence_id,system_id]].attributes[attr_id]
srm[srm_id].sequences[[sequence_id,system_id]].methods[row_id].attributes[attr_id]

// select all responses from a certain sequence invocation record
// 'response' is a core attribute
srm[srm_id].sequences[[sequence_id,system_id]].response

// select all responses for 'row_id'th row of the sequence
srm[srm_id].sequences[[sequence_id,system_id]].methods[row_id].response

// access sheets and systems involved in the SRM
srm[srm_id].sheets[sequence_id]
srm[srm_id].systems[system_id]
```

List. 12: SRMPath Notation - High-Level

languages such as Java and Python) or the bracket notation which is typically used for key-value data structures (e.g., dictionaries/maps). Essentially, we define a mix of both in order to navigate the SRM tree structure and its connected nodes in order to access their attributes.

The high-level semantics of the SRMPATH notation are exemplified by the abstract path expressions given in Listing 12. The keywords `srm`, `sequences` and `methods` refer to the core entities of the notation, here SRM actuation (sequence) sheet records and method invocation records, respectively. In order to select certain SRM sequence- or method invocation records, the brackets offer a way to specify selection (i.e., filtering) criteria, here in terms of indexing criteria (i.e., predicates) based on the unique identifiers of the entity of interest. Sequence execution records can either be selected via numeric indexing based on the matrix notation (i.e., I_{ij} where i denotes the row and j the column) or via the unique identifiers of a sequence and a system. A method invocation record is identified via its row index (i.e., row identifier) in the sequence. In order to select all entities, either no brackets or selection criteria are defined or wildcards can be used (e.g., $*$).

Core attributes of each entity are accessible via the dot notation. For example, to get the name of the current SRM selected we use `srm[srm_id].name`. Analysis attributes, on the other hand, are accessible via the dot notation for core attributes as well (i.e., `attributes`), but since those represent a collection of attribute-value pairs, we use the bracket notation to select a certain attribute via its unique identifier type.

The basic SRMPATH notation allows much more powerful path expressions to be defined. In future revisions of the SRMPATH notation, we envision much more powerful ways to select certain entities or data. For example, for this, we could realise "boolean indexing" as supported by XPATH [248] and sophisticated data analysis frameworks such as PANDAS (e.g., `srm[srm_id].sequences[[rows > 5]]` to select all sequence invocations records where the number of rows in a sequence is larger than 5). Likewise, one may apply slicing criteria as known from Python list indexing to select the first X, the last X or a certain range of entities (e.g., `srm[srm_id].sequences[0,5]` to select the first 5 sequence invocation records).

The two path expressions in Line 13 and 15 in Listing 12 demonstrate the versatility of the path notation based on two common SRM analysis tasks. Whereas in Line 13 all the responses (i.e., outputs) of a certain actuation sheet are collected, in Line 15 only the $\texttt{row\_id}^{\text{th}}$ responses are collected. Finally, for the sake of convenience, the attributes of sequence sheets (based on SSN) and systems can be accessed via their corresponding keywords `sheets` and `systems` respectively.

6.5 Data Layers

In order to obtain measures for the analysis attributes of interest in the arena, we require an efficient, scalable analysis architecture. A certain overhead introduced by static and dynamic analyses has to be "accepted" for the measurement of individual software systems (cf. Section 16.1). However, given the potential scale of SRMs, the order of magnitude of measurement overhead is several times higher, since measurements have to be created and assimilated on a potentially large set of systems that are configured in the SRMs at hand.

This not only demands efficient processing, but also demands efficient measurement "logistics" such as the collection and storage of the obtained measures. Apart from actuations and (scope-aware) measurements, these include all the execution traces (data) that are produced by analyses (e.g., call graph representations) as well as their "post-processing" products that are used to compute structured and aggregated results which can later be interpreted by users.

6.5.1 Analysis Architecture

Figure 6.5 presents a high-level overview of the observatorium's analysis architecture which has been designed to scale to meet the aforementioned needs. It is inspired by the recent advances made in data-driven processing to solve the problem of processing massive amounts of data. It is based on a distributed architecture of dedicated computing machines that are connected via a sophisticated cluster

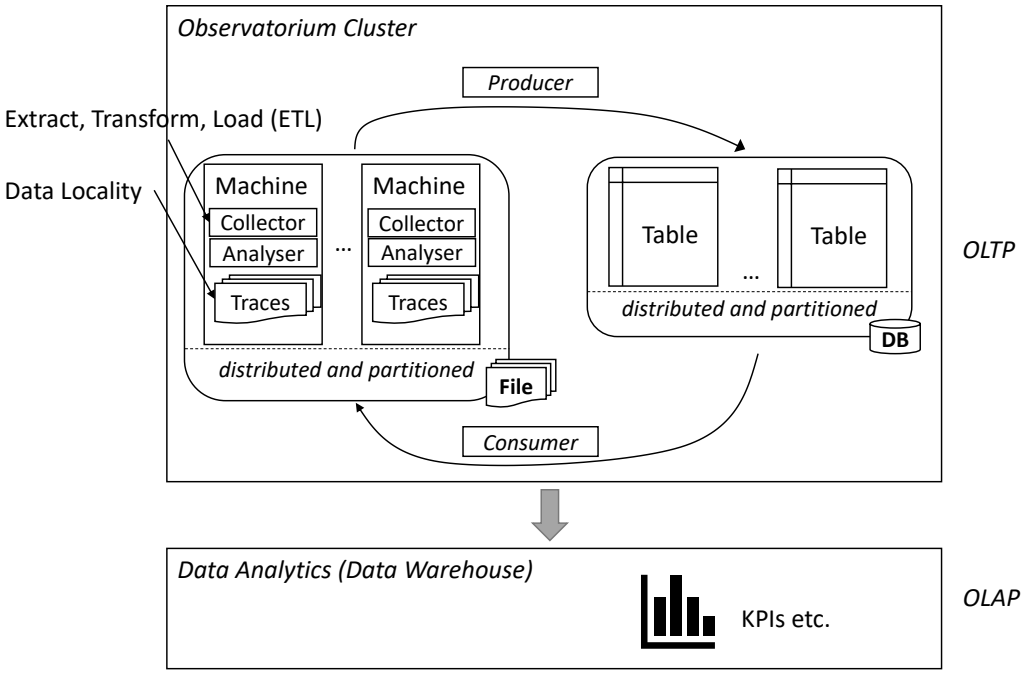


Fig. 6.5.: High-Level Overview of Distributed Analysis Architecture

middleware that run the software systems participating in the arena in parallel. A subset of systems is assigned to a local machine and are processed (i.e., built, executed and measured) in parallel by taking advantage of state-of-the-art build automation, essentially enabling "horizontal" scaling. The distributed architecture, however, takes scalability one step further and adds the capability of "vertical" scaling through a cluster of dedicated computing machines.

In summary, the analysis architecture revolves around the following established design principles of data-driven distributed architectures to meet the scalability needs for creating and analysing measurements —

- *Data Locality*: The building, execution, tracing and analysis of a software system happens on the same cluster machine,
- *ETL (Extract, Transform, Load)*: The collection of execution traces and system resources, its processing and the storing of structured data follows the well-established ETL process,
- *Event-Driven Publish/Subscribe Pattern*: Modelling and realising both local and global chains of analyses (i.e., measurement pipelines).

The building, execution and tracing of software systems is data-intensive. It involves the creation of local project builds, the resolution of required resources

(e.g., dependency artefacts) as well as the data and traces that are produced as part of the system's execution. The execution data generated as part of the measurement process of systems, in particular, can become large. The idea behind the principle of data locality is that the processing (computing) of the data happens "close" to their location (i.e., on the computing machine where the system builds are located) [104]. This principle is not only applied for execution data that needs to be processed, but also for the "structural knowledge" stored in a distributed database.

In order to support the systematic collection and post-processing of software measurements, the observatorium follows the principle of the ETL process as often used in data warehousing applications [48]. The three phases of the process are realised by the "collector" framework that runs on top of the arena in the observatorium on each distributed machine. The data obtained from the execution of a system is collected centrally in an event-driven way. Once a new execution trace is ready for processing (i.e., for extraction as part of the "extract" phase), it is picked up by the collector. Registered "analysers" can then post-process the data in the "transform" phase in order to produce structured representations of measurements. Finally, in the "load" phase, the structured representations are then stored in a distributed database. Note that locally stored execution traces can be kept on the machines that produced them. They are available for future processing through a distributed file system.

As already explained, in terms of the collector framework the ETL process is event-driven. From a high-level perspective, the event-driven mechanism follows the classic publish/subscribe pattern [243] that consists of producers that "publish" information and consumers that read and process them. To support custom analyses in a flexible way, "analysers" can "register" for new data and get notified when it is ready for processing. This principle is applied both in the local context of measurements (i.e., on each machine in the cluster) and in the remote context. Producers can store new data locally or remotely. Consumers that are interested in certain data can "subscribe" and get notified once it is ready. The use of the publish/subscribe pattern helps to automate analyses and to model simple measurements chains (i.e., pipelines), from the generation of call graphs to their processing and the final computation of measures. As a consequence, measurement chains can be constrained by scoping criteria in order to realise additional scope-aware measurements.

6.5.2 Script-Driven vs Data-Driven Processing

The distributed analysis architecture presented before defines two basic data processing layers, each of which is discussed in the following sections —

- *Online Data Processing Layer - "Script-driven" (OLTP)*: a (distributed) transactional database (and file system) to efficiently store and query observational data generated as part of analysis pipelines,

- *Offline Data Processing Layer - "Data-driven" (OLAP)*: a data analytics layer built on top of the previous layer that allows the efficient "offline" processing of observational data in terms of data analytics.

The core difference between the two layers is that analyses done in the former depend on the execution of a pipeline script that mainly stores transactional observations (i.e., hence "online"), whereas the latter analyses do not depend on a pipeline script execution (i.e., "offline"), and thus allow efficient "ad hoc" queries to potentially large amounts of transactional observation data.

The former powers all the data-related operations of the observatorium, including the representation and storage of SRMs including sequence sheets, invocation records and custom analysis attributes. The data can be stored and queried as part of LSL analysis pipelines (Chapter 10). This includes the availability of the `SRMPath` notation to allow custom queries and post-processing of SRMs during the execution of a pipeline script (e.g., test-driven filtering of systems that match certain behaviour described in terms of actuations). The latter data processing layer, on the other hand, enables advanced, large-scale data analytics [48] with sophisticated data mining tools.

6.6 Observational Transactions Processing (OLTP)

We use a relational modelling approach from classic RDBMS to represent SRMs in the "Online Data Processing Layer" of the observatorium. Today's RDBMSs are mature and use a well-defined, structured query language to manage relational data efficiently in terms of tables of rows and columns (i.e., SQL) [197, 108].

The downside of relational databases is their inflexibility with respect to the strictness of their schemata. A database schema needs to be specified in advance before data entries can be inserted. The definition of a table includes columns and their types as well as the definition of unique identifiers (primary as well as foreign keys that point to other tables) and integrity constraints. From this, it follows that the insertion of data (i.e., rows) that do not adhere to the schema is simply rejected by the RDBMS. Moreover, even though the schema can be altered at any time, this involves significant effort (e.g., migrating existing entries). Therefore, to store observational data of any kind (i.e., actuations as well as custom analysis attributes), classic relational database schemata are limited.

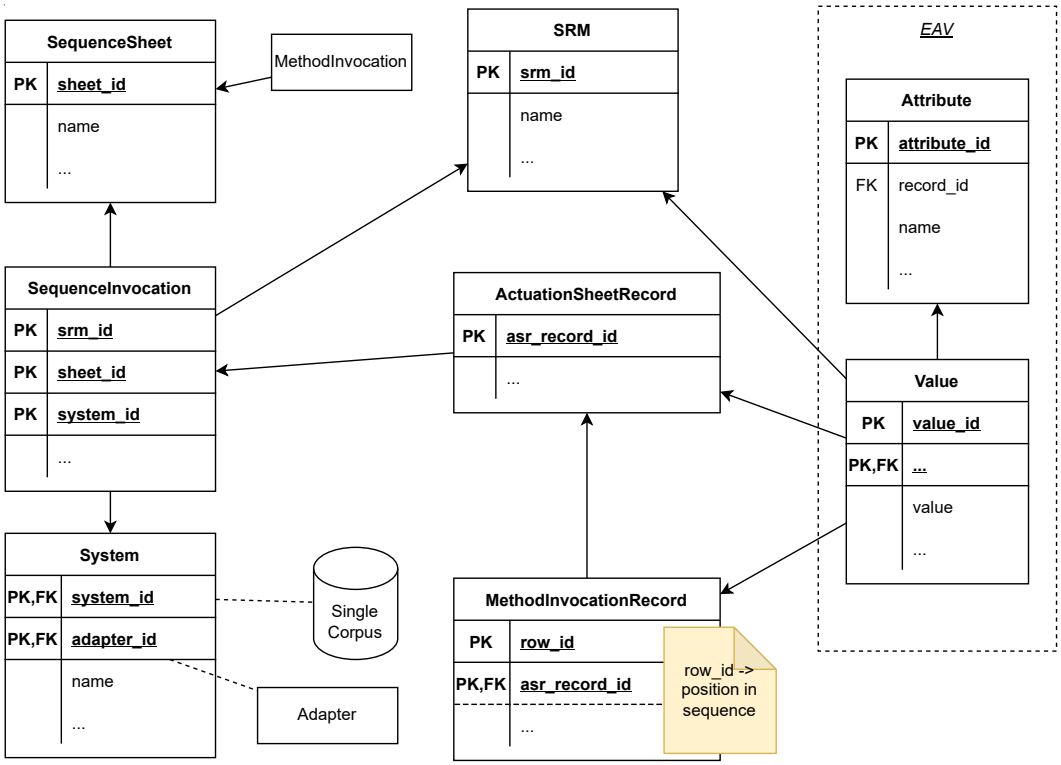


Fig. 6.6.: High-Level ER Diagram of SRM (Relational) Schema based on EAV

In order to gain the flexibility of storing arbitrary observations, we adopt a schema modelling approach that is inspired by the "Entity-Attribute-Value Model" (EAV) that allows a less strict schema (sometimes referred to as "open schema") to be realised that is both able to be strictly defined as a relational schema, but provides enough flexibility to encode analysis attributes (i.e., observational data). As a consequence, the schema of custom analysis attributes does not need to be known beforehand. Instead, new attributes are simply inserted as data entries in terms of key-value mappings. EAV has its roots in the LISP programming language where it serves as a general model for knowledge representation (i.e., information is stored as attribute-value pairs) [220].

A high-level ER diagram of the observatorium's SRM-related relational schema that applies the EAV modelling approach to support custom analysis attributes is presented in Figure 6.6.

The ER diagram provides an overview of all known entities that play a role in the data domain of an SRM based on classic database normalisation of relational data. To repeat the basic relationships, an SRM contains sequence invocations which are linked to their record table (i.e., `ActuationSheetRecord`). Similarly, to enable the white box view of SRMs, method invocations are linked to their records as well. The right-hand side of the ER diagram shows the integration of custom analysis attributes

based on EAV. Whereas the entity `Attribute` characterises a (new) attribute by its type, the entity `Value` depicts its actual value. New attributes can be inserted into the database based on the given schema by simply adding either a new attribute entry (i.e., row) in the `Attribute` table or by referencing an existing one. Apart from standard measurements, the distributed analysis architecture of the observatorium relies on this schema to store any (execution) data or references that are deemed to be of interest to users.

Note that systems that participate in the SRM in the arena may need to be adapted to the interface of a functional abstraction of interest (Section 9.3). Attributes `system_id` and `adapter_id` form a compound key and refer to a specific adapted configuration of a system.

6.6.1 Representation of Actuations

The handling of actuations (stimuli and responses) in our database schema is similar to the handling of custom analysis attributes. Technically, we use the EAV subschema to store observed actuations too. From this perspective, the entity `Attribute` becomes an `Observation` of some type, whereas the entity `Value` stores the actual output(s) of the response. The same concept applies to the storage of stimuli.

The actual representation of (object-oriented) inputs and outputs (i.e., value and type) of systems as a result of method invocations is more challenging. In order to be useful and effective, inputs and outputs need to be represented in a way that allow their efficient comparison at "query" time.

As mentioned before, since relational database schemata are strict and limited with respect to the data types used for each column of a table, one practical solution is to serialise inputs and outputs to strings (sequence of characters). The objective of string serialisation is to establish a common serialisation format that allows for reasonable (string) comparability of actuations.

In order to prevent the loss of precision, the `Attribute` table variant `Observation` is extended with a couple of additional columns including (amongst other things) additional information about —

- the value's (data) type,
- the "raw" representation of the value, and,
- "classifiers" that were assigned by users or analyses (e.g., null handling behaviour is seen as equivalent to exceptional behaviour).

The actual serialisation of information is dependent on the particular programming language used to define the systems executed. In our research prototype LASSO, we

defined a string serialisation format that is based on a JSON document representation [42]. Since the JSON document representation is extensible by its very nature (schemata may be defined as well), they can even represent cyclic structures such as object graphs.

6.7 Data Analytics (OLAP)

The database schema introduced in the previous section allows the storage and analytical processing of SRM-related records efficiently. Moreover, the SRMPATH notation can be used inside analysis pipelines in order to analyse, transform and selectively extract attributes of interest (e.g., certain responses as part of actuations).

With respect to enabling sophisticated data analytics on top of SRMs, the "Online Data Processing Layer" also has limitations, however, because it is based on the OLTP approach that is optimised for the needs of efficient transactional processing. The downside of OLTP is that advanced, complex queries as usually required by data analytics are typically too costly and too slow for large amounts of data [197].

The comparability of observations (i.e., analysis attributes) based on custom criteria is reduced in the OLTP approach, therefore, since it is less flexible. The data-driven analysis of observations (i.e., execution traces), especially the analysis of actuations from one or more SRMs typically result in large amounts of data. To visualise the potential scale, the number of stimulus/response pairs in SRMs can be approximated as —

$$\#SRMs \times \#SequenceInvocations \times \#Systems \times \#MethodInvocations \quad (6.1)$$

The second "Offline Data Processing Layer" of the observatorium, therefore, builds another data layer on top of the first layer in order to provide a "data warehouse" where complex queries can be formulated in order to analytically process SRMs and their records. This layer is inspired by the OLAP approach that is not designed for transactional processing and data consistency, but optimised for complex queries involving data aggregation as well as for feeding mining algorithms and machine learning pipelines. OLAP systems were initially proposed to support decision-makers in business intelligence. In our context, the OLAP data layer enables sophisticated data analytics of SRMs.

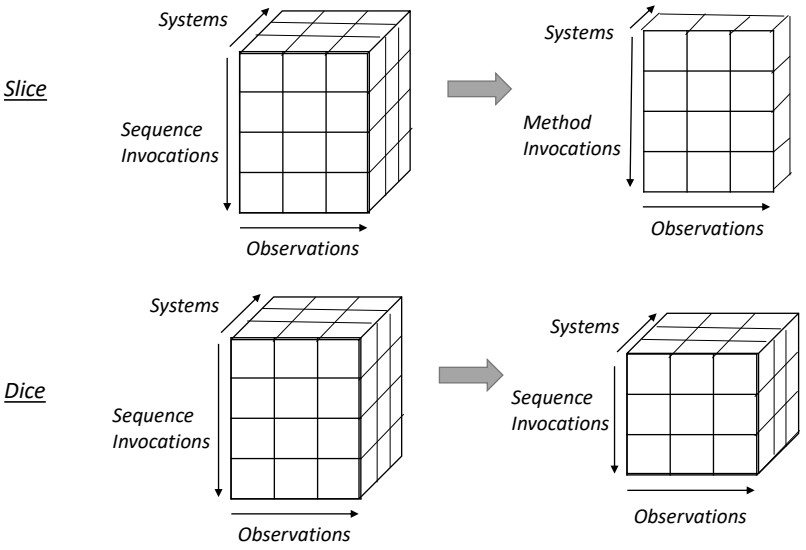


Fig. 6.7.: SRM Cube - Slice and Dice Operation

6.7.1 Analytical Operations - OLAP Cube

Conceptually, OLAP's data model is basically a multidimensional data set (i.e., multidimensional array) that can be interpreted as an extension of a relational table that supports any number of dimensions instead of two dimensions. It is also often referred to as a "cube" (or hypercube if there are more than three dimensions) [100]. A cube allows multidimensional data from multiple perspectives to be viewed and contains two basic types of tables: (1) *fact tables* that describe transactions (i.e., observational transactions), and (2) *dimension tables* that elaborate on certain attributes of a transaction.

SRMs can be naturally represented using the cube model. Here sequence- (i.e., actuation sheet) and method invocation records can be regarded as transactions that are stored in the fact table. Each record possesses a list of observations about the actuations exhibited at execution time as well as custom analysis attributes such as performance or scope-aware measurements. These can be regarded as the aspects/attributes of the transaction and thus represent dimensions. Since we consider black- and white box views on SRMs, the dimensions are also hierarchical in our case (sequence invocation level vs method invocation level).

As illustrated in Figure 6.7 and 6.8, the cube model supports four analytical operations that can be used to explore and analyse SRMs —

- *Consolidation (Roll-up)*: summarising data along one or more dimensions (aggregation and accumulation),

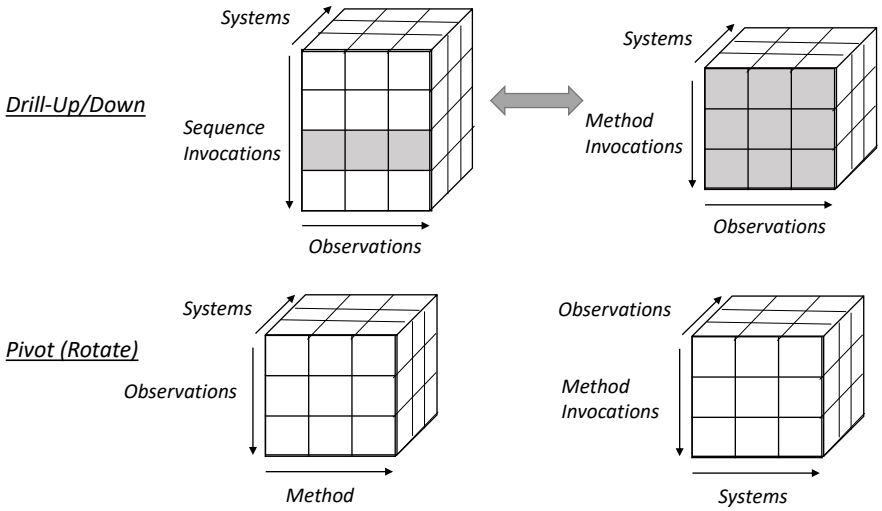


Fig. 6.8.: SRM Cube - Drill Up/Down and Pivot Operation

- *Drill-down/up*: navigating among different levels in the SRM (sequence- and method invocation level),
- *Slice and Dice*: slicing subsets of one or more dimensions of the SRM cube and filtering the SRM cube into a subcube,
- *Pivot (Rotate)*: rotating the cube to view different perspectives of dimensions.

The consolidation operation involves typical aggregation functions like `max`, `min`, `sum`, and `mean` based on certain grouping criteria (i.e., one or more dimensions). These can be used to basically describe a dimension. An intuitive example here is to aggregate performance measures obtained for sequence invocations such as execution time. In this particular example, execution time resembles a dimension at the level of sequence invocations. Using the mean aggregate function, we can return the average execution time of sequence invocations.

Drill-up/down operations facilitate navigation between the black box and the white box view in the SRM. Starting from one or more sequence invocation records that contain "summarised" observations (i.e., black box view), we can drill down to the level of method invocation records (i.e., white box view) which contain most detailed observations. Since we allow any attributes to be stored at the level of a single SRM, we may also drill-up to the SRM level that contains the most "summarised" observations.

Finally, the pivot operation facilitates the rotation of the SRM cube to view its dimensions from a different perspective. Technically, this operation relates to the two typical formats of tables – the "wide" and "long" formats. The observations

Wide Format

Method	Response	Branches
push	“”	5
pop	“hello world”	3
size	1	1

Long Format

Method	Variable	Value
push	Response	“”
push	Branches	5
pop	Response	“hello world”
pop	Branches	3
size	Response	1
size	Branches	1

Fig. 6.9.: Wide vs Long Format of Tables

of invocation records in the wide format, for example, are stored as additional columns, whereas in the long format they are stored as additional rows in the table (typically by using a “variable” column and a “value” column). As a consequence, information about a unique method invocation is not repeated in the wide format, but is repeated in long format. Changing a table from the wide to the long format is also referred to as “lengthening” or “widening” the data. An example visualisation of the observations obtained for a method invocation record is provided in Figure 6.9.

Part III

Populating the Arena

7 Creating a Single Corpus of Executable Software

Having a suitable corpus of executable software systems is vital for the envisaged observatorium, since the arena needs to be populated with systems that are "executable", and thus testable by means of sequence sheets. The required corpus should ideally contain a *large number of diverse, non-trivial, up-to-date, real-world* software systems [70, 251, 230, 24, 68, 171]. However, achieving all these properties is a major undertaking, and to date has only been achievable by curating executable software corpora by hand [4, 185]. Manual curation, however, is a non-trivial, laborious activity that consumes a lot of time and effort, and hence often leads to corpora that have three major weaknesses – they are small, rigid and/or brittle (see Problem *P1* in Section 1.2).

A natural way to reduce the need for manual curation activities is to attempt to *automate* as many of the tasks of the curation process as possible, while supporting both the evolution of corpora and their extensibility for individual analyses (i.e., other usage scenarios). This chapter explains how a single, underlying corpus of executable software systems can be created from a variety of diverse data sources. The presented approach for assimilating a corpus of executable software systems lays the foundation for an automatic curation capability that provides advanced system selection services. These are introduced in the subsequent chapters.

7.1 Software Repositories, Projects and Artefacts

Software systems and their software components can be obtained from a variety of *data sources*. Today, they are usually stored in two main data sources which we refer to as software (or code) repositories —

- *Source Code Management Systems (SCMs)* such as *Git* [93] and *SVN* [238] (Subversion) which developers use to manage and maintain source code and related resources (i.e., current and older versions of software projects) to drive software development,
- *Artefact Repositories* which are used to publish and distribute "packaged" artefacts of software applications and libraries to make them available for others

(e.g., Maven Central [217] for distributing Java artefacts and Python Package Index (*PyPI* [192]) for distributing Python packages).

We use the term *software repository* in the broadest sense possible, thus we consider any local or remote locations which contain software (including plain source code or compiled code such as Java byte code classes). This also includes any readily-available (manually) curated corpora or any other collections from which code can be obtained. For example, continuous integration systems such as JENKINS [126] can be treated as artefact repositories, since they process one or more software projects a day, thereby generating a plethora of code-related artefacts (e.g., project builds, execution traces within test reports, quality reports etc.). In a sense, question and answer websites for programmers like *Stack Overflow* [219] can also be regarded as software repositories. Many answers and solutions for technical questions contain code snippet examples as well as external references to code solutions. However, since the answers and solutions usually contain a high percentage of natural language text, such repository types are beyond the scope of this work.

Note that existing curated (software engineering) corpora may also be maintained in SCMs such as *Git* or distributed as a (local) bundle. To keep our terminology simple, we use the term software repository (or just repository) to refer to any organised collection of source code.

7.1.1 Source Code Management Systems (SCMs)

SCM repositories such as *Git* repositories usually manage a collection of software projects, which in turn may be further divided into subprojects (also referred to as submodules). Popular examples of large Open Source SCM repositories are *SourceForge* [215] that formerly relied on the SVN protocol and *GitHub* [95] that relies on the *Git* protocol.

A *software project* (or module) typically sits in an addressable sublocation of a repository[1] and depicts a bundle of project-related resources which may include code units (either compilation units or pre-compiled units), documentation, resources such as assets and configuration files, as well as build instructions (either manual descriptions or build scripts) which define and declare the build lifecycle and the (compile-time) dependencies of the project at hand. A modern SCM like *Git* transparently captures the evolution of projects by means of committing, branching and tagging models. Nowadays, software projects, especially Open Source projects, are often "forked" (regarded as an activity part of "social coding" [222]) so that they

[1] e.g., *JavaParser*'s project subpath in *GitHub* is: https://github.com/javaparser/javaparser.git

can be developed further in parallel and then finally merged. Such development practices typically result in many similar versions and variants of the same projects and systems found in different subpaths of the repository.

7.1.2 Artefact Repositories

Whereas SCMs typically maintain plain source code of software systems which must first be compiled (e.g., Java classes), either manually or automatically based on the presence of build scripts for build automation, artefact repositories such as local or remote Maven repositories (e.g., Maven Central) distribute so-called software artefacts. A *software artefact* is created from a software project (or module) and bundles a (sub)set of project resources and/or generated resources as part of certain build processes. The default type of artefact is the packaging of pre-compiled production code such as Java byte code classes[2] which are typically ready for execution. However, developers may also opt to publish additional artefacts from software projects including the plain source code, documentation and test artefacts (e.g., unit test code).

Artefact repositories offer a high degree of automation. Usually they are used as dependency resolvers to facilitate and automate the (re)use of third-party libraries as part of the software development process and software usage. Artefact repositories such as those used by the Maven ecosystem [233, 217] operate on a well-defined repository structure. They employ a repository model which defines the layout and the artefact storage model. On top of the repository model, rules, conventions as well as assumptions have been established to automate the distribution task of artefacts. Many artefact repositories work on the premise that an entire ecosystem for build automation is used to manage software projects, their third-party libraries as well as the generation of artefacts. Even though *Maven* is typically used to manage Java projects from which artefacts are deployed to Maven repositories, other build automation tools such as Gradle [99] can build and publish Maven artefacts as well, since they also support Maven's repository model and adhere to its rules and conventions.

Software projects using build automation are usually enriched with useful project-related information and structured metadata, such as information about the (transitive) libraries on which they depend and project-related properties (e.g., website authors, issue tracker as well as environmental build and execution profiles). Since build automation tools such as Maven also publish the produced artefacts in local or remote Maven repositories, the target repositories also contain such rich metadata.

[2] Note that languages like Python use an interpreter and may generate compiled code "on-the-fly".

7.1.3 Software Engineering Corpora

To date, various software engineering corpora have been created for the purpose of supporting software engineering experimentation. For this purpose, they were curated from a variety of sources (i.e., often Open Source) using largely ad hoc strategies. Some corpora contain entire software projects sourced from SCMs, but others only contain incomplete code units, with respect to missing project context, such as single classes, single methods or even code snippets. Basically, they can be divided into two groups based on their executability: (1) executable corpora, and (2) non-executable corpora.

Examples of general purpose executable corpora are XCorpus [68], 50-K [172] and NJR [185]. An example of a non-executable corpus that is used to evaluate code clone detection approaches is BigCloneBench [226].

Overall, prevailing corpora exhibit a high degree of heterogeneity with respect to their repository layouts, their contents and their degree of scriptability for building, executing and analysing the software systems contained using custom tooling (if they support these capabilities at all).

7.2 The Challenge of Diverseness

From the above description of existing software repositories it is evident that there is a high degree of *diverseness* in the used repository models, contents (in terms of software projects and artefacts) as well as scriptability and tooling support. In fact, in the extreme case, neither of the latter may even exist. Integrating existing software repositories and their software projects and artefacts into a single executable corpus of software systems, therefore, poses a major challenge. To automatically integrate new content into an executable corpus, a software repository needs to be scanned for available software projects and artefacts, then the individual file structures of projects need to be analysed in order to construct the project context and gather information about available source code, its build structure and its execution. Without knowing how a repository is organised, the analysis, identification and extraction of software systems cannot be automated, and hence has to be largely performed manually.

A further challenge is the lack of a systematic and uniform project model of software projects that provides a clear description of the project layout (i.e., contents) and its build process. Even though some projects obtained from SCMs may support build automation, these are often not guaranteed to be buildable and executable. Missing or incomplete information about required third-party libraries in a project and strong assumptions about the execution environments or the build order of modules are only two examples of hard problems that hinder the generation of

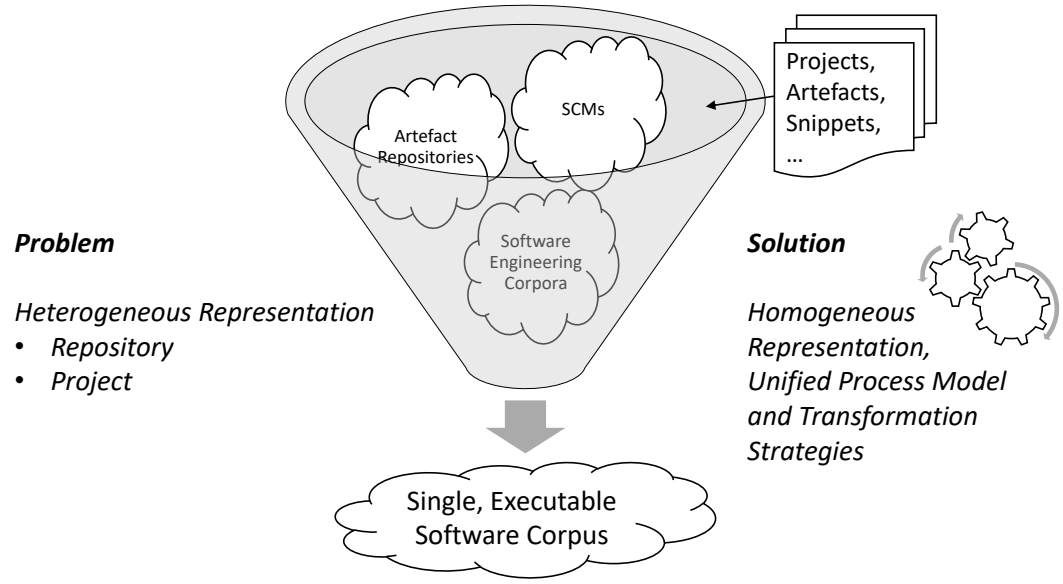


Fig. 7.1.: Corpus - Problem of Diverseness

executable software artefacts. In the extreme case, authors of projects (e.g., Open Source projects) may choose to not automate the build process at all, may use uncommon practices (e.g., different project layouts), or may not even disclose how to build a software project.

As illustrated in Figure 7.1, our solution to integrate software systems obtained from heterogeneous software repositories and projects into a single, executable corpora is to define a —

- common representation of software repositories, projects and artefacts,
- unified software corpus creation process (i.e., pipeline) which covers all the required (code) analysis steps,
- set of transformation strategies to handle the specifics of custom software repositories, software projects and artefacts on a case-by-case basis.

Based on the common representation of repositories and projects, the idea is to define a standardised analysis pipeline which acts as a "funnel" to transform existing software repositories and their software projects / artefacts into a single executable software corpus. For this, each software repository under consideration needs to be analysed on a case-by-case basis to infer its structure in order to realise an (automated) repository transformation strategy which aims to automatically integrate its contents into our unified representation. Similarly, a set of project transformation

strategies is required to detect particular project types (e.g., projects managed by a certain build automation tool) and to analyse their structure and contents in order to produce artefacts in the desired representation. More importantly, if software projects only contain plain source code, such strategies need to attempt to build them (also referred to as build script synthesis [185]). It is important to note that project transformation strategies are orthogonal to repository integration strategies (i.e., a repository may contain a variety of project types, since developers are free to choose their own).

The key advantage of the aforementioned approach is that the analyses of arbitrary repositories and their projects can be conducted systematically in a unified process while achieving high levels of automation (assuming the existence of appropriate transformation strategies).

7.3 A Single, Underlying Corpus of Java Classes

This section introduces our approach for creating a single software corpus of executable Java classes. Even though the discussion is specific to Open Source Java software, the concepts can also be applied to other (object-oriented) programming languages and internal (private) software repositories. To achieve a common representation of software repositories and Java projects, our corpus creation approach takes advantage of the existing ecosystem of the build automation tool Maven. We leverage Maven's well-defined, extensible project object model (also referred to as POM) to represent and manage Java projects of arbitrary type and structure, and Maven's repository representation (repository layout and storage model) to store and retrieve code-related artefacts for execution (e.g., Java classes). Maven's extensible project object model enables the definition of all important properties of Java projects including their project structure (i.e., location of the production code, test code, resources etc.), building profiles, environment profiles, dependency information as well as structured metadata about the project (e.g., author, versioning and license information) [195, 194].

To create a set of transformation strategies for Java code repositories and Java projects, our basic approach is to "mavenise" existing software repositories and their Java projects to analyse them, on the one hand, and to keep and store executable Java classes, on the other hand. While Maven's project object model provides a convenient way to define a project's build lifecycle and dependencies, Maven's repository layout offers a convenient and systematic way to transform existing repositories of software projects into artefact repositories with a well-known structure.

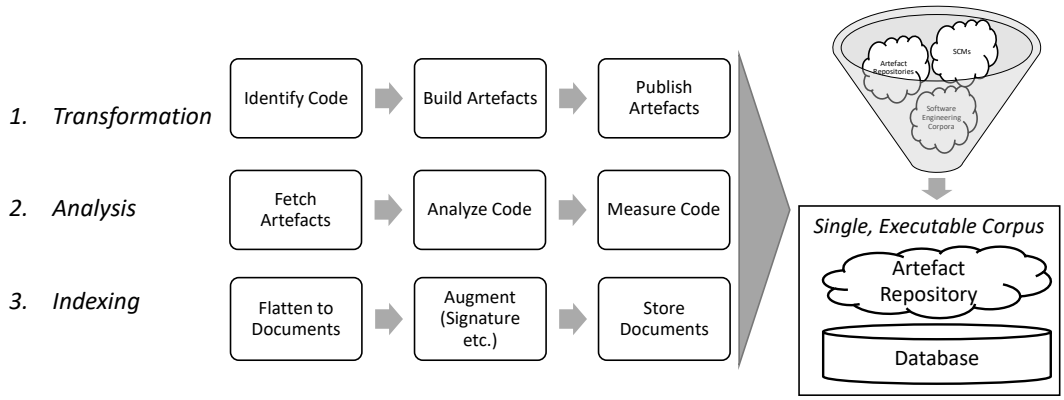


Fig. 7.2.: Unified Process Model for Executable Corpus Creation of Java Classes

Figure 7.2 illustrates the observatorium's overall process model for the automated creation of an executable Java corpus from arbitrary repositories and Java projects. It involves three main steps —

- *Transformation*: Java projects and artefacts in a given software repository are identified and then transformed into Maven artefacts that are published to the corpus's artefact repository,
- *Analysis*: each published artefact is retrieved and analysed to infer a representation of its code units and project,
- *Indexing*: a searchable database of Java classes and methods is created from the inferred representation of the code units and project.

The executable corpus itself consists of two core components, an artefact repository of executable Java systems, and a database from which Java classes can be efficiently selected. These are the key enablers for the automatic curation capability offered by the observatorium. Note that the unified process model loosely resembles the classic phases involved in the creation of a searchable index (i.e., database) for web search engines (i.e., building a full-text search engine in information retrieval [168]). Each phase of the unified process model is discussed in greater detail in the subsequent subsections.

7.3.1 Transformation

The transformation phase attempts to "mavenise" a Java repository and its contained Java code to produce Maven artefacts (i.e., packaged as jar files) which are then published to the corpus' Maven artefact repository. If the source repository is already

a Maven artefact repository (e.g., Maven Central), or it contains packaged artefacts, obviously this step can be skipped and only the artefacts need to be published into the target Maven repository.

Apart from this special case, the first step of the transformation stage involves the "crawling" of a code repository to download its contents (either fully or partially based on some selection criteria). Then all the Java projects and artefacts it contains need to be identified. If the given repository has a custom repository layout or uses a custom storage model, a transformation strategy first needs to be defined to scan and detect Java projects and artefacts. After detecting a set of Java projects, if the project is not already managed by Maven, the next step is to "mavenise" the project by synthesising a Maven project object model. To do this the file structure needs to be scanned for code (plain source code as well as compiled code), possible third-party libraries and all the relevant resources needed to build and execute the Java code.

Depending on the particular code repository and Java project types, additional metadata about the code repository and/or project in the project object model may be stored in the Maven repository. This can also be analysed in the second phase to put more searchable information into the database. Once a set of Java projects has been successfully detected and transformed into Maven artefacts, they are published into the artefact repository of the executable corpus. The default mode is to create and publish several Maven artefacts from a Java project. By default, we produce artefacts that contain the compiled code, the plain source code, and if available, the test classes of the Java project.

7.3.2 Analysis

After a set of Maven artefacts has been produced and published, the role of the second phase is to fetch and analyse them in order to detect how the code units are represented. In the third phase these units are then stored in a database to enable the efficient selection of code units (i.e., retrieval of classes and methods). The goal of the analysis phase is to create a rich representation of the code units found in the fetched artefacts. This facilitates the subsequent selection and filtering of Java classes and methods.

The analysis starts with the scanning of all Java classes, then proceeds with the parsing of each Java class detected using an abstract syntax tree (AST) [1]. The objective of AST analysis is to dissect each class to collect all its declared methods (i.e., method signature and method bodies). Each class and method is described and enriched by its object-oriented properties. This includes properties obtained from type hierarchy analysis, inheritance tree analysis as well as dependency and

invocation analysis. Apart from direct properties obtained from the source code, contextual information is collected in order to model relationships between classes and between methods (e.g., method `A` has owner class `B` etc.). The representations of classes and methods are further enriched with the structured metadata extracted from the project object models which are shipped as part of the artefacts.

Finally, in order to further characterise the code units under analysis, we store additional software metrics like size-based complexity metrics, and also hash the source code of class bodies and method bodies to detect code duplicates [206].

7.3.3 Indexing

The goal of the last phase is to store the code representations of classes and their methods inferred in the second stage for later efficient retrieval. In other words, the third stage lays the foundation for the basic curation capability of the executable corpus.

To store the code representations, they are first "flattened" to key-value based documents. The database (i.e., index structure) is realised using SOLR/LUCENE [234]. SOLR is a web-service and search platform built on top of LUCENE's search engine library. It offers full-text indexing and NLP querying capabilities. LUCENE supports classic NLP techniques such as tokenisation, stopping and word stemming. By default, it computes textual relevance judgments using BM25 (tf-idf) [168]. Our basic approach effectively combines the static analysis of Java code using ASTs with the NLP techniques offered by SOLR/LUCENE, since the properties derived by static AST analysis are made searchable using NLP techniques.

Figure 7.3 shows a high-level overview of the schema that we use in order to store all the properties derived in the second stage in terms of key-value fields in documents. It basically models projects, artefacts, classes, methods, their properties as well as their relationships.

Technically, the schema translates the code representation into two types of documents: (1) class documents, and (2) method documents. The relationships between classes and methods are retained and modelled using unique identifiers. The information and relationships of projects, artefacts, measurements and metadata are "flattened" into key-value pairs for each class and method document.

After the documents have been created from the code representation, they are augmented with additional fields. Based on common practices in NLP, some data items in the documents are stored multiple times in different fields in order to improve the retrieval of classes and methods. Examples include the creation of fields for similarity matching and the creation of fields for equivalence matching. Most importantly, the augmentation step creates multiple fields which contain variations

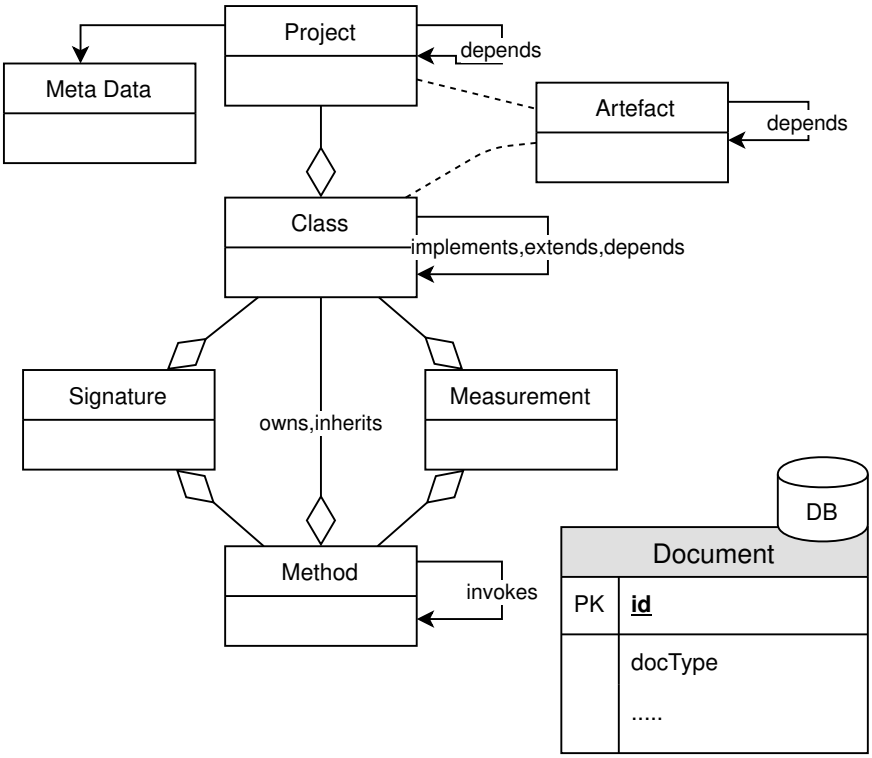


Fig. 7.3.: Database Schema - High-Level Overview

of method signatures to support flexible querying strategies for interface-driven code search which is explained in the next chapter (Chapter 8). Overall, the aforementioned techniques help to improve retrieval performance in terms of recall.

7.4 Executable Project Builds

The main motivation for building an executable corpus is to obviate manual curation tasks by automatically executing, stimulating and observing the behaviour of software systems in the observation arena. Even though the executable corpus lays the foundation for executing classes and methods textually selected from its database (i.e., via NLP-driven querying techniques) based on all the build information in its containing project, it still cannot guarantee that they can be executed. Because of the many (technical) hurdles that occur in practice such as wrong assumptions about versions and execution environments, and missing or unresolvable artefacts, the underlying executable corpus only partially solves the execution challenge.

The goal, therefore, is to make the classes and/or methods textually selected fit for execution, by attempting to automatically build them and to automatically resolve their dependencies. In other words, this step resembles classical tasks in

build automation and tries to automatically configure and set up executable "builds" of each class/method and all its required execution context.

7.4.1 Setting up Project Builds

The execution of a particular class and/or method on a computing platform (e.g., JVM) requires a project build to be set up. Since our executable corpus is based on the Maven ecosystem, a default Maven project is populated with the following resources by default —

- a synthesised build script that describes the current class/method and declares the containing Maven artefact as a dependency,
- the given test sequences to stimulate the class/method.

The synthesis of a Maven project object model (i.e., build script) facilitates build automation for each candidate retrieved from the database and allows the builds to be managed systematically. As we will see later (see Chapter 12), Maven-managed project builds allow the building and execution of builds in parallel to enabling vertical scalability (taking advantage of multi-threading capabilities on a single machine) as well as horizontal scalability (distributing parallel builds on a distributed cluster of machines).

The build script of each candidate build can be configured in a fine-grained way to express constraints and assumptions about the execution environment (e.g., Java version compatibility). Moreover, build scripts offer great extensibility with respect to the configuration of readily-available "plug-ins" (e.g., measuring code coverage as part of test execution).

7.4.2 Executing Project Builds

Due to the fact that project builds of classes/methods are managed by Maven and that the executable corpus is built on top of the Maven ecosystem, a dependency resolution mechanism is available right from the start. For each class/method the corresponding Maven artefact which is referenced through Maven coordinates in its model retrieved from the database (i.e., key-value fields in SOLR document), is retrieved from the corpus' artefact repository. Because of the transitive way dependent artefacts are declared in Maven projects, the dependency resolution mechanism also automatically retrieves all transitive artefacts on which the artefact depends. Sometimes, it is not possible to execute a generated Maven build, either because its dependency resolution failed (e.g., transitive artefacts cannot be resolved)

or the classes stimulated at execution time failed to load due to class resolution problems. In these cases, the selected classes/methods and their corresponding builds are typically rejected by default for the remaining steps of the process.

8 Text-Based Software Selection

The single, executable software corpus described in the previous chapter possesses all the basic properties needed to support custom selection criteria for distilling data sets of executable Java systems that exhibit a certain set of properties of interest. The observatorium's "text-based selection" capability represents the database layer which lays the foundation for querying classes and methods based on the key-value fields stored as part of the corpus creation process (Section 7.3). In other words, it provides the capability of a code search engine supporting textual queries. Since this does not involve the execution of classes and methods, the determination of functional behaviour is limited to the strengths of the applied NLP-driven techniques (i.e., mainly matching of names and interface signatures). Test-driven selection (see Chapter 9) takes into account the true behaviour exhibited by software systems, but it builds on, and hence requires, text-based selection techniques to retrieve a preliminary set of candidate systems.

8.1 NLP-Driven Selection of Software Systems

The emergence, at the turn of the century, of large-scale, Open Source software repositories accessible over the Internet and efficient NLP-driven, full-text search tools such as LUCENE to index and analyse their contents, spawned a range of software engineering tools referred to as "code search engines" [212] or "code recommendation systems" [204]. Their main aim is to help facilitate software (code) reuse [149] – that is, the finding of existing code units that match the needs of a new software system.

Code search engines and code recommendation systems often go hand in hand and basically only differ in the way users interact with the search technology. Code search engines essentially require users to perform a proactive search by creating some kind of explicit "query" whereas code recommendation systems typically do not require users to be so proactive but "suggest" potentially useful reuse candidates to them based on observations of their work. However, the latter usually rely on the former in order to carry out basic searches over large populations of code units. Well known examples include the CODEGENIE recommendation tool [154] driven by

the SOURCERER [17] search engine and the CODECONJURER [120] recommendation tool driven by the MEROBASE [117] code search engine.

Since code is a form of semi-structured, text-based data [4], the majority of dedicated code search engines are essentially based on full-text search. However, the query languages of most code search engines assign a special meaning to the core components of source code such as methods, classes etc. This can extend to the level of allowing users to specify the interface signatures of the functional abstractions they are looking forward in terms of one or more method signatures. The goal of such interface-driven queries is to find all classes and methods that implement the specified interface.

NLP techniques suffer from the same limitations as static program analysis approaches in that they do not execute source code to identify its true behaviour. Retrieval techniques based on NLP techniques, however, work surprisingly well to obtain a first set of candidate systems that exhibit the desired properties. For instance, interface-driven code search built on top of classic NLP techniques exploits the fact that software systems that realise similar functional abstractions probably have similar interface signatures in terms of their names and input/output types.

8.2 Keyword and Filter Queries

The simplest way to select software systems of interest is to use the keyword query capability of full-text search engines like classic keyword queries from popular, general-purpose search engines such as Google. In the observatorium, SOLR queries operate on the key-value fields that are stored in the corpus' database of class and method documents (Section 7.3.3). They allow fields storing certain properties about classes and methods to be matched via similarity or via equivalence. Keyword queries can be combined with boolean queries containing several sub-queries combined via boolean operators.

Intuitively, classes or methods can be matched by their name assuming that the name of a class corresponds to the functional abstraction it implements. Consider the following keyword query matching a single field "name" (using SOLR's query syntax) —

```
name:"Base64"
```

In this example, we retrieve class documents which have a similar name to "Base64" (see functional abstraction Base64 in Section 2.2). This query returns an ordered list of class documents, ranked by SOLR's relevance score which is computed based on the degree of similarity of the matches. Assuming that classes called "Base64"

actually implement Base64 functionality, true implementations of Base64 might be returned. Alternatively, since the plain source code is stored in the database, it is also possible to assume that classes which frequently contain the term "Base64" in their source code may be actual implementations of Base64. But even in this case, we suffer from the same limitations as in the first case. The desired functional abstraction can only be specified by one or more keywords in a generic way.

Keyword queries can be augmented by filter queries that define additional constraints. Filter queries use the same syntax as key word queries, but they act as a filter on the retrieved class/method documents and do not affect the relevance score of the matched documents. For our purposes, filter queries may be used on virtually all key-value fields that are used in class and method documents. First, we use filter queries to select the code unit of interest (i.e., class or method). Filter queries are also useful for filtering classes or methods based on object-oriented properties or measurements. For example, the following filter query selects classes which have at least 10 lines of code and which implement the Java interface `java.util.Collection` —

```
type:"class"
interface:"java.util.Collection"
loc:[10 TO *]
```

Like keyword queries, filter queries also support boolean matching as well as range queries (as demonstrated by the `loc` filter that defines a range starting at 10 with no ending criterion).

8.3 Interface-Driven Code Search (IDCS)

Keyword queries as well as filter queries offer a basic approach for retrieving classes and methods. They are limited, however, since functional abstractions can only be described in a generic way (i.e., by name or concept).

An approach that is built on top of keyword and filter queries, but considers the specification of functional abstractions, is interface-driven code search (IDCS). It works on the premise that classes and methods that implement similar functionality (i.e., functional behaviour) likely exhibit similar structural properties in terms of their interface signatures [61, 141]. The observatorium therefore supports an interface-driven search capability inspired by the MEROBASE query language (MQL) [117]. In our case, IDCS queries are formulated using LQL, the "LASSO Query Language", which realises the interface notation introduced in Section 4.1 that is used for sequence sheets.

To query software systems (i.e., Java classes) which potentially implement the stack abstraction, for instance, the following IDCS query can be specified in LQL —

```
Stack {
    push(Object)->Object
    pop()->Object
}
```

In order to execute the query, it is first translated into a lower-level SOLR query. Based on the parsing tree of the IDCS query (built from LQL's query grammar), parts like the naming and types are translated into corresponding key-value subqueries as well as filter queries to retrieve class matches from the database (i.e., single, executable corpus). In addition to class queries, IDCS also supports method queries and optional filters. The filters are specified as classic SOLR filters. A method query for the Base64 encoding abstraction, for instance, can be specified as follows —

```
Base64 {
    encode(byte[])->String
}
```

It has the same format as the class query, but the abstraction/class name is optional. In this example, the sought-after method signature is called `encode`. It accepts a single input parameter of type `byte[]` (i.e., a byte array) and returns a value of type `String`.

8.4 Improving Relevance

The relevance of matches attained through text-based selection technologies has a major impact on the relevance property used to distil data sets. In the following subsections, we discuss three alternative ways to further improve the relevance of the selected software for the process of creating data sets.

8.4.1 Improving Recall

Depending on the nature of the functional abstraction described in an IDCS LQL query, the recall of classes or methods matched by their signature might be low, since there may be many mismatches of parts of the method signatures (i.e., naming or type mismatches). This issue of signatures mismatches can be traced back to the vocabulary mismatch problem [87]. Since different developers may name functional abstractions differently, and may use different types to design classes and methods, signature mismatches occur frequently. To address this problem and to improve recall, the observatorium employs the following optimisations to maximise signature matches —

- tokenising names and types,
- storing multiple variants of signature representations in terms of key-value fields (cf. [117]),
- allowing query expansion of names based on synonyms and antonyms (using a thesaurus based on WordNet [177]),
- allowing query expansion of types based on their type hierarchy.

While the first optimisation is performed at index creation time, the remaining optimisations are done at query time. It is worthwhile noting that all optimisations can be realised in either of the two ways (cf. [168]).

The idea of storing multiple variants of the same signature is a technique proposed by Hummel for the interface-driven code search capability of the MEROBASE code search engine [117]. Based on a set of heuristics, frequently occurring mismatches of signatures are accommodated. These types of mismatches include the ordering parameters and their types and variants that either include or exclude the name of a method.

While translating an IDCS query to SOLR subqueries, several subqueries are constructed which target different variants of the signature as defined by fields in the database. Basically, all those signature-related subqueries are combined in order to match more classes or methods. In our approach, all the signature variants proposed for the MEROBASE code search engine are retained and additional useful variants are added to provide even more matching flexibility. These include the generation of signature variants which include types in two formats: short and long. The long format represents types based on their fully-qualified name, whereas the short format also stores their simple name (e.g., `java.lang.String` vs `String`). In this case, we can match a specific type by its fully-qualified name or any type by its simple name to return a larger result set. Moreover, we also generate a signature variant which encodes the number of input parameters. All these signature generation strategies contribute to better and more flexible matching of classes and methods.

At query time, we employ three additional optimisations in order to improve recall. Firstly, we apply tokenisation based on the camel case format typically used as a Java code design style to name Java classes and methods. Class and method names are split by camel case and for each constituent part, a corresponding subquery is created (e.g., `encodeBase64` is split into `encode` and `Base64`).

Secondly, we apply two automatic query expansion techniques [45] to further improve recall for IDCS. With respect to naming terms defined in IDCS queries, the expansion strategy attempts to limit the vocabulary mismatch problem by generating

a variety of SOLR subqueries from the names extracted from the signature. It performs query expansion by looking up synonyms and antonyms in a WordNet-based thesaurus [177], where method and class names are assumed to be interchangeable. Optionally, this strategy can be combined with tokenisation of names. Similarly, signature types are expanded to their compatible types identified from either their type hierarchy or from a list of predefined compatible/convertible types.

8.4.2 Increasing the Diversity of Matches

One of the most desired properties of a data set is that it contains a set of diverse software systems (i.e., in terms of the classes and methods they contain). At the present time, however, large software repositories frequently contain large numbers of code duplicates [164]. These are created for a variety of reasons like (copy and paste) software reuse [212], forking strategies in social coding practices etc. A data set that exhibits a high level of simple redundancy in terms of code duplicates may lead to non-generalisable results depending on the purpose at hand (e.g., software experimentation).

The observatorium incorporates two approaches to improve the level of diversity in distilled data sets. First it introduces a basic capability to identifying code duplicates based on weak indicators such as code hashing and size-based software metrics at query time. Second, as explained in Chapter 10, it incorporates more sophisticated code clone detection techniques and tools which can be used as part of custom analysis pipelines.

To identify and reject code duplicates (or clones) as part of IDCS, we take advantage of the additional properties stored about classes and methods in the database. Code clones can be detected and rejected based on common content hashes computed at index creation time. If the source code of a pair of classes or methods is identical, only the first retrieved class or method is kept in the list of matches. An alternative way to identify code clones is to compare two classes or methods based on a set of properties. For example, size-based software metrics can be used to weakly indicate that a pair of code units are probably identical. Theoretically, this approach works for all prevailing properties stored in the database like equivalent naming (e.g., identical fully-qualified class names).

8.4.3 Sorting Matches

Matched classes can be sorted (i.e., ranked) based on their relevance. By default, class/method matches for IDCS are sorted based on the default scoring computed by SOLR using the BM25 matching algorithm [202]. It is computed in terms of a

single criterion that considers the similarity of returned matches to the original query. Depending on the purpose of a data set, more advanced ranking schemes which are tailored to specific needs can be used. These include single-criterion ranking strategies based on the selection criteria available in the observatorium as well as multi-criteria ranking strategies such as SOCORA [142, 143].

The sorting of matches is particularly useful in analysis scenarios in which the objective is to find a relevant match as quick as possible. High quality ordering increases the likelihood that a more relevant match is closer to the start of the list, making the search for relevant matches more efficient (e.g., putting more relevant candidates nearer to the front of the list).

9 Test-Driven Software Selection

The software selection approaches described in the previous chapter are entirely based on text-based queries, including keyword-driven searches and interface-driven code searches. However, this means that, regardless of how sophisticated they may be, software systems can only be retrieved based on their "purported" behaviour (i.e., the behaviour they appear to exhibit based on the identifiers in their source code or text in their comments) rather than based on their "true" behaviour as defined by the algorithms built into the source code (Problem *P2* in Section 1.2). To improve the precision of matching results, therefore, the observatorium incorporates an additional "behavioural sampling" mechanism, referred to as "test-driven selection", that leverages the data structures introduced in Chapter 4 and 6, and the arena. This chapter describes the aforementioned "test-driven selection" technology developed for the observatorium.

While test-driven selection can increase the precision of searches for specific system behaviour (i.e., by reducing the number of false positives), it can also reduce the recall (i.e., by reducing the number of true positives) due to interface mismatches between the candidate systems and the sought-after functional abstraction. In order to increase recall in the test-driven selection of software systems, it therefore incorporates an advanced adaptation approach that aims to adapt none-matching interfaces to a common "interfaces" by synthesising adapters. This is inspired by the well-known adapter pattern of the "Gang of Four" (GoF) [90].

9.1 Behaviour Sampling

Behaviour sampling techniques [191] apply the idea of classic software testing to the selection of software systems from software repositories. Here software systems are selected by observing their actual behaviour and matching it to the desired behaviour of some functional abstraction.

Test-driven code search engines (TDS) [119], in particular, exploit the idea of behaviour sampling and combine it with test-driven development (or test-first development) from agile practices (cf. [33]). Such engines allow the matching of software systems from software repositories by their true behaviour through the

specification of suitable test sequences (e.g., unit tests) derived from the specified behaviour (set of actuations) of the sought-after functional abstraction. The hypothesis underpinning behaviour sampling is that a few test sequences are sufficient to characterise the behaviour of software systems with high confidence [191, 61, 141].

Unfortunately, most of the test-driven code search engines developed to date were research prototypes that are no longer maintained, and none of them were specifically tailored to support the focused analyses entailed by "big code". In order to match the behaviour of functional abstractions with satisfactory confidence there are, however, two fundamental challenges that need to be overcome. The first challenge is the *execution logistics* involved in obtaining run-time observations of the behaviour of a large set of software systems retrieved from software repositories under controlled conditions. This requires a systematic definition approach for stimuli as well as for the recording and collection of responses that scales to the needs of big code. State-of-the-art approaches, however, rely on classic testing tools like unit test frameworks that do not meet these needs (Problem *P4* and *P5* in Section 1.2). The second challenge is the *idiosyncratic choices* made by software engineers when designing the structure of a software system to realise a certain functional abstraction. This results in a plethora of software systems obtainable from software repositories that realise the same or highly similar functional abstractions, but in different ways. To resolve interface mismatches, a promising approach is to adapt candidate software systems to a common "interface" by synthesising adapters as explained in the second part of this chapter.

The observatorium's test-driven selection approach tackles the first challenge of execution logistics using the arena data structures introduced in Chapter 4 and 6. The test-driven selection of software systems is realised systematically and follows a strict separation of concerns with respect to the process steps involved. Figure 9.1 provides a high-level overview of the three basic steps involved in the test-driven selection process which are discussed in greater detail next.

9.1.1 Population

The first step of the test-driven selection process needs to (a) specify the behaviour of a functional abstraction of interest, and (b) preselect a set of candidate systems that match the specified behaviour. Our approach achieves this through a combination of sequence sheets that specify the behaviour of the desired functional abstraction.

Stimulus Sheets

Since a small number of test sequences are usually sufficient to retrieve a set of systems matching a specified behaviour [190, 141], users of the observatorium need to

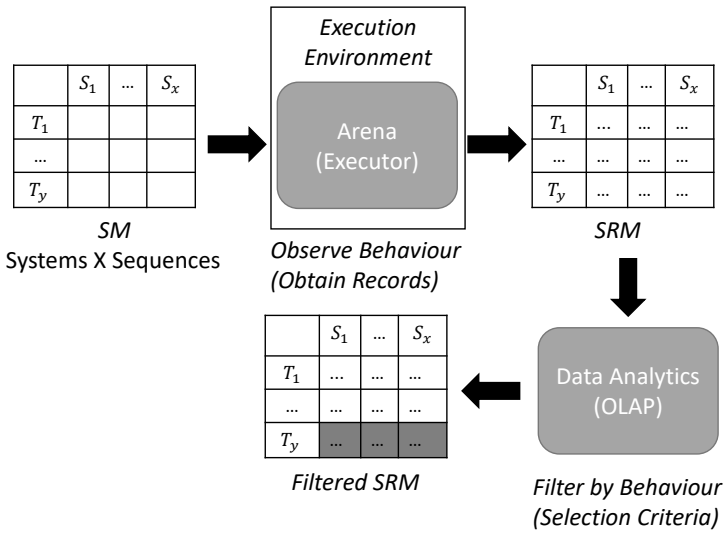


Fig. 9.1.: Test-Driven Selection Process - High-Level Overview

generate one or more actuation sheets that describe "typical" and unique actuations of the functional abstraction. Alternatively, if an existing system is available as a reference implementation (i.e., oracle) to define the expected responses, users only need to define stimulus sheets.

In classic unit testing there are two basic ways to "generate" test sequences: (1) they can be defined *manually* by engineers, or (2) they can be generated *automatically* using a test generator tool (AUTG) such as RANDOOP, based on random testing [184], or EVOSUITE, based on meta-heuristics [174] (see Section 16.4.2). Another interesting option for big code approaches that operate on large software repositories is to reuse existing test sequences in the same way as software systems. Since large software repositories are used to manage the entire lifecycle and evolution of software projects, text-based selection can be used to mine existing test classes and methods [123, 199, 241, 24]. The mining of test sequences is orthogonal to manually-written or automatically generated tests. Both types of test sequences can be mined from repositories.

The sequences described in Chapter 4 support all the aforementioned strategies for obtaining test sequences to populate the arena.

Software Systems

The search space of software systems is usually huge considering the wealth of software components collectable from large software repositories like GitHub or Maven Central. Since the execution of large combinations of sequences and systems is resource and time intensive, it is important to narrow down the initial search space

of software systems to those that are considered of interest with respect to certain preselection criteria. Fortunately, the structured database of code and code-related properties in the executable corpus offers text-based querying capabilities. In theory, we can use any query types like keyword searches to narrow down the number of software systems of interest (sometimes referred to as speculative searches [117]). However, the majority of the usage scenarios for the observatorium assume either functional abstractions specified manually or represented by a given reference software system. Since a functional abstraction defines a set of interfaces, a natural approach to obtain a preselection of software systems from the executable corpus is to use IDCS as explained in Section 8.3.

Here the required target interfaces of the functional abstraction of interest can either be specified explicitly in terms of the interface notation (either manually by users or based on a reference software system that realises a certain functional abstraction), or they can be inferred from the given set of sequences that is used in a later step to stimulate the list of software systems matched by the text-based selection step. Technically, the latter option is simply an intermediate step that attempts to extract the interfaces of the functional abstraction(s) of the software system under test from the sequences.

Sometimes test-driven selection is used to facilitate and support intermediate analysis steps in the arena as part of larger analysis pipelines (Chapter 10) that are executed by the observatorium. In this case, the preselection is performed in earlier analysis steps that map and reduce a collection of systems based on individual criteria.

9.1.2 Observation

Once the arena has been populated with stimulus sheets and software systems, the arena executes all the former on all the latter to obtain observational records of the behaviour exhibited by the systems. Since the execution of sequences and systems takes place in a sandbox execution environment (Chapter 12), observations are made under controlled, environmental conditions in order to ensure the comparability of systems's behaviour.

The observational records are stored in the resulting SRM as explained in Chapter 6. Optionally, the observation step may also involve the measurement of additional (scope-aware) software metrics, or the collection of additional tracing information (e.g., call graph analysis) in order to obtain and collect more data points for the sake of post-analyses.

9.1.3 Filtering

The step of filtering software systems is performed by comparing the exhibited behaviour of systems to the desired behaviour. Traditionally, test-driven search engines use strict selection criteria to select what systems to filter out, such as full functional equivalence. In other words, classic test-driven search engines reject candidate systems from the result set if just one of the actuations for the same stimulus are different (Section 3.4.1).

In order to apply the aforementioned selection criteria, the SRM produced by the arena is analysed in the data analytics layer (Chapter 6.7) provided by the observatorium based on its white box view by conducting equivalence checking of the stimulus/response records stored in the actuation sheets. This analysis identifies which of the candidate software systems exhibit different behaviour to the sought-after functional abstraction.

Since behaviour-aware selection in the observatorium is performed in a data-driven manner based on tabular representations of actuations, the filtering step is completely decoupled from the observation step (in contrast to existing behaviour sampling approaches.). This gives users more control over the selection process, since they may specify their own custom selection criteria (e.g., less strict criteria). This is why we refer to test-driven code search simply as test-driven selection of software systems to highlight the fact that the approach is not limited to the strict filtering of software systems based on full functional equivalence.

9.2 Code-Driven Selection

Test-driven selection requires the user to specify the desired behaviour of some functional abstraction in the form of test sheets. However, in principle it is also possible for a user to specify the desired behaviour by providing an exemplary implementation of a functional abstraction (i.e., an executable specification).

For the main software reuse scenario of test-driven code search, this may at first seem counterintuitive, since what is the point of searching for software systems that implement a particular functional abstraction if one is already available. However, since the production and testing of high-quality, trustworthy implementations is much more effort than creating prototype ones, so called code-driven (or code-to-code search [144]) can still be cost-effective.

Large-scale analyses for big code and experimentation in software engineering go way beyond the main objective of software reuse (i.e., ideally matching a high-quality reuse candidate to save time and efforts [115]). For example, code-driven selection can be used to explore the level of (behavioural) redundancy in repositories

to judge core properties like sparsity [4], to drive studies about behaviour sampling techniques (or code search engines) [141], or to support automated test generation approaches that are driven by diversity (i.e., heteromorphic redundancy) (see Chapter 14).

Test-driven selection can be extended to support code-driven selection as long as effective test sequences can be generated automatically. Instead of using user-defined test sequences to stimulate and judge the acceptability of textually selected systems, observation-based, code-driven selection approaches use automatically generated test sequences and judge the acceptability of the retrieved systems by comparing their exhibited behaviour (i.e., responses) to the reference implementation defined by the user.

In effect, therefore, observation-based code-driven selection in the observatorium is built on top of test-driven selection. By using AUTG, the arena can be populated with sequence sheets generated from a reference implementation. The remaining population, observation and filtering steps are exactly the same. In Chapter 14, we introduce several analysis pipelines, some of which realise test-driven selection in a code-driven way.

9.3 Adaptation

Until now, we have assumed that software systems have the exact interface that the stimulus sheets in the arena expect. However, the recall achievable by test-driven search would be drastically reduced if only candidates with exactly matching interfaces were considered. As mentioned at the beginning of this chapter, therefore, the first challenge in behaviour sampling is to overcome interface mismatches between the systems and functional abstractions of interest. If mismatches can be resolved, the recall of test-driven selection can be significantly enhanced.

In traditional software engineering projects, the goal is to create a system realisation whose behaviour is equivalent to, or subsumes, a specification of the sought-after functional abstraction. However, if reusable software systems already exist it may be possible to adapt them to implement the desired behaviour. This can be achieved by mapping the methods in the interface of the desired behaviour to the methods of the existing software system.

Before we explain our approach to adaptation, we first establish basic terminology for adapted software systems based on the formal model introduced in Chapter 3.

9.3.1 Adapted Software Systems

The definition of behavioural subsumption introduced in Section 3.4, and thus the notion of "implements", assumes that the interface of a functional abstraction, f, is an exact subset of that of software system s (i.e., with the same names and the same formal parameters occurring in the same order etc.). However, when software reuse is considered, it can be the case that a functional abstraction f, and a software system, s, do not share any interface in this strict sense, and therefore s cannot be regarded as an implementation of f. However, it is possible to achieve such a relationship by adapting the method signatures in the interfaces of s to the method signatures in the interface of f.

A software system is referred to as an **adapted system** if it is produced by adapting interfaces in this way. Formally, an **adapted system** is composed of an adapter, a, and a base system, b (i.e., the pair (a, b)). An **adapter** is conceptually a software system which offers an interface, and maps invocations of the operations contained in this interface to methods contained in the interfaces of another system. The mapping $m_f : (a, b) \rightarrow f$ defines a mapping m_f of the adapted system (a, b) to functional abstraction f. The behaviour of an **adapted system**, s, composed of an adapter, a, and a base system, b (i.e., $s = (a, b)$), is the set of all possible actuations of the (a, b) pair when invoked via the interface offered by a.

In the simplest case, it is possible for the adapter of an adapted system to be "empty". This occurs when the adapter simply offers the same interface as b, or a subset of the interfaces of b, and trivially maps the methods in the interface of a to those in the interfaces of b. While such an empty (or null) adapter is of no value in practice, it is useful for theoretical completeness of the formal model. At the other extreme, it is possible that system b and all its interfaces can be adapted by a, but b is "empty". In this case, the full behaviour resides in the adapter (i.e., adaptation behaviour), so theoretically the original system is not of any value, but the adapted system, however, delivers the required behaviour.

Note that since adaptions are usually made for a purpose (i.e., to implement a functional abstraction), adapted systems are usually implementations of particular functional abstractions. Next, we explain and discuss the synthesis of adapters to adapt systems to a required interface.

9.3.2 Adapter Synthesis for Java Systems

The choice of technology for the synthesis of code-based adapters for software systems is basically tied to the capabilities of the object-oriented programming language used. Even though we discuss our adaptation approach in the context

of Java, the concepts can be transferred to other programming languages such as Python as well.

The synthesis of adapters for software systems built with modern object-oriented programming languages can be realised in two different ways. *Compile-time adaptation* encodes the adapter logic into classic code within classes and methods (i.e., source code) which then need to be compiled. The adapter logic basically implements a certain interface of interest and delegates invocations to the adaptee. Compile-time adapters are essentially applications of the GoF adapter pattern [90]. The adapter pattern[1] is a design pattern for converting the interfaces of one system to the target interface of another system. This allows clients of the software systems to use their preferred interface.

In contrast, *run-time adaptation* postpones the decision to intercept, inject and even synthesise methods of classes to match a certain interface specification to run-time. To do this, the underlying programming language is required to support the run-time instrumentation and introspection of classes and methods (i.e., to support meta-programming capabilities [161]).

Both ways of synthesising adapters for classes have their advantages and disadvantages. Whereas run-time adaptation determines adaptation bindings on-the-fly, no source code needs to be generated that needs to be compiled as in the former case of compile-time adaptation. However, to date, many tools in the Java ecosystem rely on the presence of compiled-classes, so the use of adapters synthesised from *run-time adaptation* is limited in those tools. Some (analysis and measurement) tools even do their own instrumentation and introspection at run-time, which may lead to collisions with adapters generated at run-time.

The LASSO prototype, therefore, realises both forms of adapter synthesis. While run-time adaptation is realised using Java's meta-programming capability, called Reflection [160], Java's compile-time introspection mechanism, is realised by synthesising classic Java adapter classes using code generation based on ASTs.

In the following, we explain the run-time adaptation approach to adapter synthesis that is applied to Java classes matched to a target interface. The compile-time adaptation approach is not discussed in greater detail, since its adaptation mechanism is akin to run-time adaptation. Likewise, sequence sheets which are used to model test sequences in the arena also require adaptation. Here again, the inner workings of adaptation are identical, so the adaptation approach discussed next applies to them as well.

[1] It is also known as the "wrapper" or "decorator" pattern.

9.3.3 Approach

Depending on the structure of the interfaces and the adaptation strategies applied, there can be multiple mappings (i.e., bindings) between the interfaces of a Java class and a desired interface. A major obstacle in adapter synthesis, therefore, is to cope with the sheer number of mappings (i.e., adapted software systems) that can be identified between interfaces [249]. The more methods and parameters that can be matched, and the more adaptation operators that are applied, the higher the number of computable mappings (sometimes the number of possible mappings may grow exponentially).

A large number of computed mappings lead to much higher execution costs, since each computed mapping depicts an adapted software system that needs to be executed and stimulated in isolation with possibly a large set of sequences. Adapter synthesis can therefore be treated and formulated as an optimisation problem. In the context of the behaviour-aware selection of software systems, the challenge is to find the optimal mapping in the search space of possible mappings that exhibits the desired behaviour of the functional abstraction of interest.

In contrast to MEROBASE's "brute-force" approach [117] which tried every possible mapping, a promising and more efficient strategy to keep the search space of adapted systems manageable is to prioritise the execution of adapted systems. The idea here is to prioritise the execution of those adapted systems that are likely to be implementations of the functional abstraction. Together with a suitable search budget (i.e., search timeout), this solution is both practical and scales to the adaptation of many software systems.

Our adaptation approach basically builds on the idea of a prioritisation schema with assigned time budgets and synthesises adapters between the interfaces of a Java class and a target interface based on three main ingredients —

- a well-defined *adaptation protocol* to communicate between adapters and their adaptees (based on meta-programming),
- a *best effort prioritisation algorithm* to cope with the sheer number of possible mappings (i.e., combinatorial explosion of the search space) between the interfaces of the adaptee and the target interface,
- an extensible set of *adaptation operators* that realise custom adaptation strategies in order to identify suitable mappings between the target interface and the interfaces of the adaptee.

Figure 9.2 shows a layered architecture diagram that illustrates the different layers of the adaptation approach to create adapted software systems for behaviour-aware

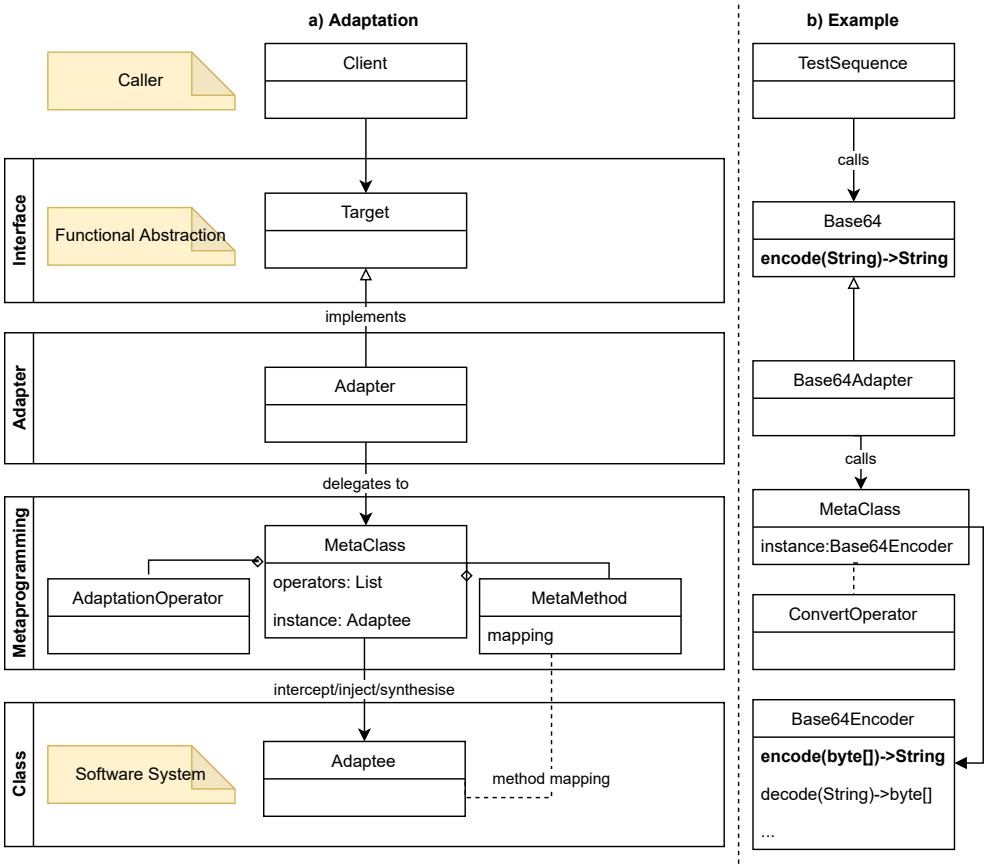


Fig. 9.2.: Run-time Adaptation based on Meta-Programming (i.e., Java Reflection) (a) - Base64 Encoding Example (b)

system selection. Alongside the architecture the figure shows a small example of an adapted system based on our running Base64 example.

The adaptation architecture for classes has four fundamental layers overall. The basic objective is to make it possible for a "client" (i.e., caller) to call one or more methods of a class instance (i.e., the adapted software system) through the target interface of some functional abstraction which resides in the "interface layer". In test-driven selection, the client is the stimulus sheet which manages an instance of a sequence (of method invocations) that calls one or more methods once or repeatedly. The adapter located in the "adapter layer" does not provide any adapter logic on its own. Instead, it just realises the target interface and forwards any methods calls to the "meta-programming layer".

The meta-programming layer is driven by a metamodel of classes and methods that represents the class properties of the adaptee. The metamodel is used to resolve possible method mappings between the target interface and the interface of the adaptee class. Using this metamodel, the meta-programming layer is capable of —

- intercepting method calls between the client and the adaptee,
- injecting additional mapping logic (i.e., managing configurations of how the adaptee's methods can be called),
- synthesising new methods and fields for the adaptee (e.g., providing implementations of default methods like equivalence checking or missing methods).

Based on these design choices, the meta-programming layer represents a well-defined adaptation protocol to enable the communication between the client, adapter and the adaptee. An extensible list of adaptation operators is applied to define and realise custom adaptation strategies to identify suitable mappings between the method signatures of the interfaces. The adaptation process and algorithm is discussed in greater detail below.

Method Resolution

The input to the adaptation algorithm is (a) the adaptee's (Java) class, and (b) the target interface consisting of a name and a list of method signatures depicted by our interface notation. The aim of the first preliminary task is to derive a structured metamodel of the adaptee class's available methods (and other object-oriented properties) as well as a metamodel of the method signatures of the target interface. The results are then used to drive the adaptation process to compute mappings between the adaptee's interfaces and the target interface.

The actual method resolution strategy employed is driven by Java Reflection. Apart from inspecting the adaptee class's declared methods and initialisers (i.e., constructors), it also inspects the entire inheritance tree of the class (i.e., all its (transitive) super classes). By default, the resolution strategy only considers subclass implementations of methods. As a consequence, any methods that are overridden in the inheritance tree of some subclass are ignored.

Having resolved all suitable methods of the adaptee class and all method signatures of the target interface, the next step is to inspect the input- and output parameter types of all the method signatures obtained.

We build a detailed type hierarchy model of each method (signature) parameter (input/output) by walking up the inheritance hierarchy of a type to find assignable types up to the root type (here `java.lang.Object`). The type hierarchy is enriched with possible type castings, relaxations (e.g., `int` to `double`) as well as compatibility (e.g., primitive wrapper types can be substituted by primitive types). Even though the type hierarchy stores the (original) position information of parameter types, it normalises them in a way that allows adaptation operators to pick any ordering of input parameters of a certain method signature (i.e., any possible permutation of a

list of input parameters). Finally, the type hierarchy model is stored as part of the models constructed from the adaptee's class and the target interface.

Adaptation Operators

Finally, the method signatures and type hierarchies obtained from the adaptee class are "matched" with the method signatures obtained from the target interface. Inspired by the strategy pattern [90], an extensible list of adaptation operators is applied that realise custom adaptation strategies inferred from coding practices and experiences (e.g., design patterns and best practices). The input of the adaptation operators are the models that represent the interface to be matched. The output of the operators, on the other hand, are computed mappings between the target interface and the adaptee. In other words, each computed mapping depicts a different configuration of how the methods of the adaptee are called, so each mapping depicts an adapted system that is subject to execution and stimulation.

Based on the role of classes and objects used in object-oriented programming, the observatorium applies two groups of adaptation strategies in terms of operators. First, programmers of a class can control how an instance (i.e., object) of the class can be obtained. There are a variety of established strategies to create an instance of a class, including the definition of (custom) constructors and the use of classic design patterns such as the "singleton" and the "factory" pattern [90]. We refer to this group of adaptation operators as "producer operators", since they return an instance of a class. Second, all the other operators that judge classic methods are part of the "method operators" group.

Table 9.1 summarises the various adaptation operators the observatorium supports in order to create an instance of a class (i.e., producer operators) as well as to map method (signatures) between the adaptee and the target interface (i.e., method operators). Note that some operators compute mappings for both class instance producers and methods. This subset of operators actually represent the intersection of those operators that operate only on the type hierarchy, since they do not take into account the actual characterisation of a "method" (either as an initialiser or normal method).

Prioritisation Schema

For classes and target interfaces of non-trivial complexity, applying a list of adaptation operators often results in a huge list of computed mappings. The number of combinations (i.e., mappings) is roughly affected by the number of matching methods, their (parameter type) permutations (position as well as compatible types), the number of method combinations (if more than one method is required), and

Tab. 9.1.: Adaptation Operators for Java Classes

Operator	Group	Description
Default	Producer	call default constructor
Null	Producer	create object with “null” values
Harvest	Producer, Method	call subset of actual parameters with harvested values (e.g., constants)
Static	Producer	static methods only, no instance required
Factory	Producer	identifies “factory” methods
Singleton	Producer	identifies “singleton” objects
Cast	Producer, Method	cast types
Convert	Producer, Method	convert from types
ByReference	Method	try “pass by value”
ByValue	Method	try “pass by reference”
Delegate	Method	identify delegation methods

the number of “producers” (i.e., instances) identified. Furthermore, the number of operators and their possibly combined action increases the search space of possible mappings even further.

The observatorium’s strategy for managing huge search spaces is based on the idea of prioritising mappings that are likely to be of higher relevance to influence the order in which mappings are executed. Based on the “closeness” of mappings in terms of their method configurations in the type hierarchy of the adaptee class, a prioritisation scheme is applied to prioritise those mappings that are most likely to match the target interface (and the behaviour of the functional abstraction of interest), thereby improving performance and scalability of the adaptation process.

The ranking-based prioritisation scheme applied to the generated list of mappings (i.e., adapted systems) is primarily based on type closeness, the location of the matched methods in the inheritance hierarchy of the adaptee’s class, and optionally, the naming of the desired interface signatures. A weighting scheme is used to compute the ranks of method permutations, and finally, based on simple parameter-tuning heuristics, the maximum number of mappings (and hence the number of adapted systems executed and stimulated) is capped.

9.3.4 Example

In order to illustrate our adaptation approach, part (b) of Figure 9.2 shows an example based on our running Base64 (encoding) example. Consider the test-driven search use case. Here a sequence attempts to call method `encode(byte[])->String` of the adaptee which was textually matched by IDCS (i.e., `Base64Encoder`) through the method `encode(String)->String` as defined by the target interface of the

functional abstraction. Since the two method signatures do not match, the adaptation approach attempts to identify a suitable mapping. After resolving all methods and parameter types of `Base64Encoder`, the available list of adaptation operators is tried. In this particular example, there are several feasible mappings. It is possible to apply the "Convert" operator (cf. Table 9.1) that converts a value of type `String` to a byte array (`byte[]`) and vice versa. Ignoring the name of the methods defined by the adaptee class, it is obvious that both `encode` and `decode` are possible mappings.

The prioritisation schema of the algorithm puts the `encode` method first, because it also takes into account the "closeness" of the naming of two methods based on textual similarity. In this case, the name of the method offered by the adaptee corresponds to the name of the method required by the target interface of the functional abstraction.

The "correctness" of the possible mappings can only be determined by the client by executing the current mapping (i.e., via the set of test sequences). Fortunately, the ordering of the ranked list of mappings returned by the schema puts the likely better match first, so the `encode` method of the adaptee is tried and executed before the `decode` method. Depending on the goal of the test-driven selection, if all test sequences pass (assuming they verify the behaviour of the mapping as well), no additional mappings need to be tried.

Part IV

Using the Observatorium

10 Pipeline Definition Language

The last missing piece of the puzzle is to integrate the aforementioned capabilities into a usable platform. In this chapter, we introduce a domain-specific language, the LASSO Scripting Language (LSL), that seamlessly integrates the aforementioned capabilities into a usable platform. LSL makes it easy for users to define and apply multistep analysis pipelines for the dynamic analysis of large numbers of software systems.

Having addressed Requirement *R1* to *R4* in the previous chapters, LSL addresses Problem *P5* (Section 1.2) by providing a dedicated pipeline definition language. This chapter, introduces the LSL language and its capabilities.

10.1 Domain-Specific Languages

A domain-specific language (DSL), as its name implies, is a declarative or imperative computing language that is tailored to the needs of a particular application domain, in order to help users address domain-specific problems more efficiently [79]. In contrast, general-purpose programming languages such as Java and Python are designed to be used in any application domain in order to solve general problems. A DSL may be realised using a customised, static language specification (e.g., a custom language grammar) or it may (dynamically) extend existing (general-purpose) languages. The former is usually referred to as an *external* DSL that relies on an independent language toolkit (i.e., parser, compiler etc.), while the latter is usually referred to as an *internal or embedded* DSL that is derived from the syntax of the host language (i.e., uses the toolkit of the host language).

A popular example of a domain-specific language that defines its own custom language grammar is the "Structured Query Language" (SQL) designed to query and manage data stored in relational databases[1] efficiently. Other popular examples include declarative languages such as XML and JSON. Modern general-purpose programming languages such as Groovy [62] and Kotlin [125] that extend the Java language, on the other hand, come with official toolkits that support the creation of custom DSLs. Since custom DSLs extend the underlying language in which they were specified, they usually inherit their syntax as well. A popular example of these

[1] Note that SQL is no longer limited to relational databases, it is also widely used as a "standardised" query language to access other data sources such as "NoSQL" databases [60].

is Gradle's DSL [99] to define dynamic build scripts or Jenkins DSL [126] to define steps of continuous integration pipelines.

The underlying principle of this thesis is to provide users with a single, domain-specific language to define observatorium applications involving the large-scale dynamic selection, analysis and comparison of software. The established terminology and associated models, as well as the proposed approaches, are therefore used to characterise the problem domain of the DSL. This allows users of the observatorium to focus on the definition and enactment of custom analysis pipelines. This chapter describes the scripting language offered by LASSO – the LASSO Scripting Language(LSL).

10.2 Scripts

LSL is based on the general-purpose programming language Groovy, which is a popular JVM-based language supporting the creation of custom, embedded DSLs that run as scripts in the execution environment. Since Groovy runs on the Java virtual machine (JVM), the scripts are executed in the same run-time environment, but in an isolated execution context. The host application can communicate with the script via well-defined bindings, thus enabling the symmetric sharing of state in terms of objects. This allows the observatorium to control the execution of LSL scripts and to seamlessly interact with them in order to deliver all required static and dynamic analysis services.

Technically, Groovy scripts are translated into Groovy classes that compile to classic Java class byte code. Scripts can be created and loaded dynamically at run-time, but their source code can also be statically manipulated using Groovy-specific AST analyses in order to enrich and/or validate them. Groovy inherits the statically-typed language property of Java, but can also be used as a dynamically-typed language in which explicit types can be omitted[2]. In the context of designing custom DSLs, dynamic-typing provides more flexibility since actual types need not be known in advance. Scripts are therefore "agnostic" of newly introduced types which facilitates the integration of existing tools and techniques. However, there is the usual trade-off to be made, since dynamically-typed languages open up a new category of errors with respect to type-mismatches (e.g., developers may intend to call a method on an object that is not of the assumed type).

LSL is an efficient declarative as well as imperative DSL with a minimal set of commands to support the usage of the observatorium. LSL's core features are inspired by other DSLs for project build management (e.g., Maven project object

[2]In fact, Groovy internally simply represents any object as the "object" type from which all objects inherit in Java/Groovy.

model and Gradle DSL), continuous integrations systems (e.g., Jenkins Pipeline DSL) and mining like BOA [75]. Since LSL scripts are based on Groovy, and Groovy in turn is based on Java, creators of LSL scripts can use the Java syntax as well as the Groovy syntax to define their analysis pipelines.

Conceptually, LSL and its commands are based on a simple domain model derived from the concepts described in the previous chapters along with a simple, but effective, workflow model of the actions that can be performed on them. *Actions* represent reusable and composable analysis steps with a well-defined life cycle. The intent behind actions is to abstract from the complexity of the underlying tasks and provide a flexible means of combining and nesting analysis steps. Even though there is no strict classification of action types with respect to their level of granularity (i.e., actions may also execute compound actions), we basically distinguish between actions that —

- *select* software systems from the executable corpus,
- *observe/analyse* the behaviour of software systems, including static analyses as well as dynamic analyses.

While the declarative features of LSL facilitate the straightforward definition of pipelines that consist of well-defined data flows in terms of analysis steps (i.e., actions), the imperative features of LSL facilitate the definition of custom analysis logic. LSL's design strives to reach a good balance between the human-readable and maintainable definition of work- and data flows, and a flexible definition of custom analysis steps (i.e., actions).

Technically, the declarative features of LSL allow the observatorium to control the execution of actions to realise the principle of "inversion of control" (IoC)[3] [91]. This, in turn, basically enables the observatorium to manage the life cycle of action executions. Well-known examples of the use of the IoC include dependency injection frameworks such as the Spring framework [245] in Java. IoC allows the implementation of actions to basically be "decoupled" from executions. Action developers can therefore focus solely on the implementation of the required steps and only need to be aware of their life cycle (i.e., via a predefined contract).

From an architectural perspective, the design principle behind LSL scripts can also be regarded as applying the publish/subscribe model where actions publish SRM-related data to a queue of some sort. Other actions can then subscribe for certain types of SRM-related data and receive notifications when a change occurs. The IoC principle and publish/subscribe architectures both enable greater flexibility

[3] Sometimes it is referred to as the "Hollywood principle/law: Don't call us, we call you".

with respect to the scalability of LSL script execution, since they allow tasks to be offloaded to a set of distributed machines.

10.2.1 Structure of an LSL Script

Since the structure of LSL analysis pipelines is inspired by the programming paradigm of data-flow programming [129], an LSL script depicts the flow of data as a series of actions. In our case the data that flows from one action to another is typically represented in the form of SRMs. Each action serves as an analysis step that can manipulate the data flow. Figure 10.1a shows an abstract example (i.e., skeleton) of an analysis pipeline written in LSL.

To execute a script, the observatorium's workflow engine (see Chapter 12) evaluates the LSL script and extracts two directed, acyclic graphs (*DAGs*) in which the edges identify important relationships among action nodes. The first DAG, referred to as the *dependency DAG* (see Figure 10.1b), is created to identify `dependsOn` relationships between all the actions represented by their `action` blocks. Based on the first DAG, the second DAG, referred to as the *execution DAG* (see Figure 10.1c), is constructed to define the order of action executions (i.e., consecutive analysis steps in the pipeline). As in other data-flow programming languages, actions can be "chained" together so the state and results created by one action can be, consecutively, further processed by other actions.

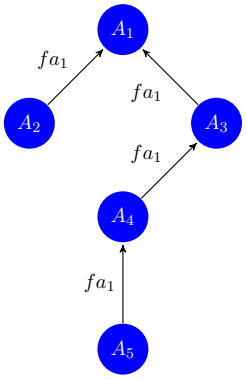
In this particular illustration, five actions are defined within a study block. The *study block* depicts the boundary of a focused analysis with a particular purpose. Action *A1* depicts the first analysis step defined by the study whereas action *A5* depicts the last step that is defined and executed.

Action *A1* is special in this case, since it defines a new functional abstraction using the `abstraction` block. A functional abstraction depicts a named "data container" that provides a single point of access to the context of a set of systems, sequence sheets and observations (i.e., records). As depicted by the domain model in Figure 10.2, the container structure wraps configurations of systems and sequence sheets contained within SMs and keeps references to the resulting SRMs that capture observations (Section 6.3). In this case, the abstraction defined by the first action is called *fa1*. Note that in the subsequent sections we demonstrate how those container structures can be populated and filled with SMs and SRMs.

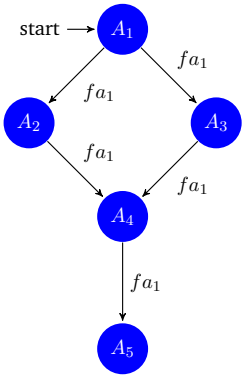
Actions can be either producers of abstractions, consumers of abstractions or both. In order to consume one or more abstractions, an action can establish a "dependsOn" relationship to one or more other actions and can selectively define the abstractions of interest from those using the `includeAbstractions` command. In fact, the dependency DAG shown in Figure 10.1b is constructed from this information. It

```
study(name:'studyName') {
    action(name:'A1', type:'A1') {
        abstraction('fa1') {
            // ...
        }
    }

    action(name:'A2', type:'A2') {
        dependsOn 'A1'
        includeAbstractions 'fa1'
    }

    action(name:'A3', type:'A3') {
        dependsOn 'A1'
        includeAbstractions 'fa1'
    }

    action(name:'A4', type:'A4') {
        dependsOn 'A3'
        includeAbstractions 'fa1'
    }

    action(name:'A5', type:'A5') {
        dependsOn 'A4'
        includeAbstractions 'fa1'
    }
}
```

(a) General Structure of an LSL Script



(b) Dependency DAG of (a)



(c) Execution DAG of (a)

Fig. 10.1.: LSL Script and Corresponding DAGs

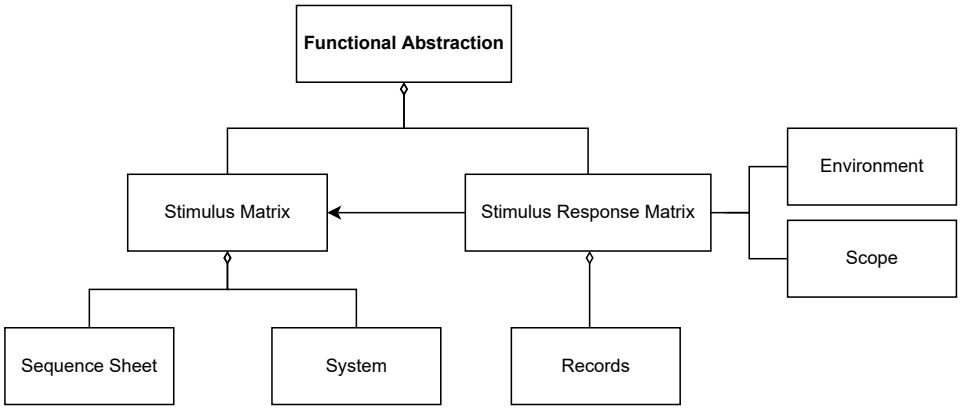


Fig. 10.2.: Functional Abstractions as Data Flow Structures in LSL Pipelines

represents the dependencies and the abstractions that are consumed for each action defined in the study block.

10.2.2 Execution

The execution DAG as illustrated in Figure 10.1c is computed from the dependency relationships depicted by the dependency DAG in Figure 10.1b. The execution DAG stores information about the order in which action nodes are executed.

This DAG is mainly used by the workflow engine to reason about optimised execution strategies in terms of optimal job scheduling and load balancing (i.e., task assignments in a distributed cluster of machines) to enable scalability optimisations (Chapter 12). For example, a simple execution strategy that can be inferred from the dependency DAG is that action *A2* and *A3* can be (theoretically) executed in parallel (vertical scaling), since they do not depend on each other.

The DAG representation can also be queried within LSL scripts and within available LSL actions to (dynamically) reason about other actions (nodes) in terms of predecessors and successors in order to guide the decision-making processes within analyses.

10.2.3 Immutability and Restoring States

In order to maintain data consistency in a pipeline of actions, the state and hence the abstraction structures produced by each action, are immutable to other actions. Instead of the original container structure, consuming actions receive a copy that can be manipulated for their own purposes.

Immutability is a key concept to achieve data consistency in the execution of pipelines in a scalable, distributed architecture. As a side effect, immutability also

allows previous computed states to be restored. Restorable execution points are beneficial, since they allow —

- data consistency to be maintained, since subsequent consumers (i.e., actions) operate on copies of abstraction containers (including SMs and SRMs),
- execution strategies to be built that scale and prevent "side effects", since actions can run in parallel when they operate on copies,
- historical data flow to be tracked,
- existing (intermediate) analysis results to be cached,
- script execution to be resumed at intermediate states (e.g., resuming at a certain action that links to the result of an existing action),
- the results of (sub)analyses to be reused at certain points in time.

Based on a standardised URI scheme, users of LSL can write actions that point to results of past executions of actions in other scripts. Using the path notation `script:study:action`, an action can depend on the state (i.e., abstractions) produced by an action of another script that was executed prior to the study script at hand. For example, an action defined in a different script can depend on the abstraction *fa1* defined in the example script by using the URI `myScript:studyName:A1`. This requires script names (i.e., identifiers) to be globally unique. Study names as well as the chosen action names must be unique within the local context of an LSL script. Conflicts between the names of actions that use the same name for functional abstractions are resolved by using fully-qualified names based on the dot notation, with the action name appearing as the prefix of the abstraction name.

10.3 Actions

The heart of LSL pipelines are the actions (i.e., analysis steps) that can be chained together in order to create and manipulate SMs and the resulting SRMs created as part of arena executions. The general LSL structure of action blocks and their most important properties (i.e., contained subblocks) in study blocks is presented in Listing 13.

10.3.1 Action Block

Action blocks define two formal parameters: (1) a unique identifier (i.e., a name that is unique in the context of a defined study block), and (2) an optional type. The

```
action(name:'ACTION_NAME', type:'ACTION_TYPE') { // type is optional
    configuration { ... } // configuration block (may be omitted)
    dependsOn 'ACTION_NAME_1', 'ACTION_NAME_2', ...
    includeAbstractions 'ABSTRACTION_NAME' // filter abstractions
    includeSequences '*' // filter sequences
    includeSystems { abstractionName -> { ... } } // filter systems

    profile('PROFILE_NAME') { // execution profile for the arena
        scope('SCOPE_NAME') { ... } // scope-definition (measurements)
        environment('ENVIRONMENT_NAME') { ... } // execution environment
    }
    abstraction('ABSTRACTION_NAME') { ... } // create new abstraction
    execute() { } // execute block

    whenAbstractionsReady() { ... } // post-processing after action execution
}
```

List. 13: Structure of an LSL Action Block

unique identifier names are used to select and match actions in the pipeline, whereas the type of an action refers to a predefined action to be instantiated. The LASSO research prototype provides a list of available actions which users can choose from (see Section 12.5.2), some of which are covered in the remainder of this section. If no type parameter is provided in the action block a so-called *plain action* is initialised that exhibits "no behaviour" by default. Plain actions can be used by LSL developers to process data using LSL commands in intermediate steps of the pipeline. Actions that operate on predefined types extend the default behaviour of existing types.

Configuration Block

The configuration block inside an action block is used to initialise an action with further parameters (i.e., key-value initialisations). This is particularly useful for initialising predefined actions with configuration values or initialising local parameters that are useful for further processing data within subblocks of the action. As in general-purpose programming languages, such parameters can be assigned to global parameters (i.e., references) that are defined somewhere in the preamble of the LSL script. This allows variable assignments to be reused. For the sake of convenience, the surrounding configuration block can be omitted, although sometimes it is necessary to define the configuration block explicitly.

The next two DSL commands, `dependsOn` and `includeAbstractions`, have already been discussed. To recap, the former is used to establish relationships between actions and accepts one or more URI schemata as introduced before, whereas the latter acts as a name filter to only select those abstractions (containers) of interest.

The SMs and SRMs shipped with the selected abstraction containers can be prefiltered based on the configured sequences and systems. Similar to the filtering of

abstractions by name matching, command `includeSequences` serves as a convenient way to filter sequences of interest by name from the selected abstractions and their SMs and SRMs. The DSL command `includeSystems`, on the other hand, provides an imperative way of filtering systems in the abstraction container via their attributes (e.g., class or package name or project-related properties).

Profile Block

The `profile` block allows users to set custom profiles for the execution environment that the arena uses for the execution of sequences on the systems. Moreover, this block also enables users to set custom scope definitions that are used to determine the boundary of systems in terms of their software components to enable well-defined measurement scopes (Chapter 5). As illustrated in Figure 10.2, the records (i.e., observations) stored in the resulting SRMs are always dependent on a particular (default or custom) execution profile. Users may be interested to observe the behaviour of systems based on a particular Java version, for instance. The environmental profile can be used to set a custom Java run-time (see Section 12.3). Likewise, custom measurement profiles can be set via custom scope definitions. Users may either use predefined scope definitions like "class-level" measurements that exclude third-party libraries or define custom scopes. One simple way to define scopes is to provide black or white lists.

Abstraction Block

As explained before, actions can either be consumers of abstractions or producers of abstractions (or both). Using the `abstraction` block, new abstractions can be defined. Note that abstraction blocks are special, since users can define their own "logic" to create a new abstraction, or they can rely on pre-existing abstractions that are produced by the particular action that is defined as the type. In the next subsection, we will demonstrate this case based on the predefined `Select` action that provides a default implementation producing an abstraction container based on IDCS.

Execution and Post-Processing Blocks

One life cycle operation of an action has already been introduced, its initialisation block `configuration`. Next to the initialisation of an action in LSL, there are two other important life cycle blocks of actions that are controlled by the workflow engine of the observatorium: (1) `execute`, and (2) `whenAbstractionsReady`. Depending on the type of action, `execute` has two main purposes. If the action is not initialised

with a predefined action type (i.e., is a plain action), this block acts as the main behaviour when the action is executed. As explained before, plain actions in LSL are typically used to do some intermediate processing of abstractions like post-processing, summarisation, measurement or even merging SMs or SRMs. If the action is initialised on a predefined action type, then the `execute` block is executed before the core behaviour of the predefined action. So in this particular case, the block acts as a pre-processing step.

Finally, the `whenAbstractionsReady` block realises the post-processing of action executions in which SRMs may be analysed and post-processed. The lifecycle blocks typically realise the processing in an imperative way based on a minimal set of DSL commands that provide access to the data that flows from action to action. Details of the most important LSL commands are introduced next.

10.3.2 Setting up Pipelines

In order to explain the construction of analysis pipelines in LSL using actions and LSL commands we use a real pipeline based on the running example of the stack abstraction.

Listing 14 presents a pipeline that realises the test-driven filtering of stack implementations based on a sequence sheet written in LSL commands using SSN (see Section 4.2). Note that in this case the sequence sheet also contains manually defined responses (i.e., expected behaviour of the stack abstraction in terms of output values) in the first column of the defined sheet. Moreover, the script takes advantage of dynamic typing, so types that are not explicitly provided are marked using the `def` keyword of Groovy[4].

Overall, the pipeline script defines a single study that consists of two actions. The first action performs an interface-driven code search on the executable corpus based on the specified interface of the desired stack abstraction (Section 8.3). The second action receives the abstraction created by the first action and configures an SM that consists of a single test sequence represented as a sequence sheet and the systems returned by the first action. Then it executes the configured SM in the arena in order to obtain an SRM that contains all the records of the behaviour exhibited by the systems. Each action is discussed in further detail below.

Selection from Executable Corpus

The preamble (i.e., header) of the script defines two important ingredients. The `dataSource` command instructs the workflow engine to set a default data source

[4]The `def` keyword is also used to distinguish between variable declarations of the local and the global script scope.

```
dataSource 'mavenCentral2020'
// interface of a Stack in LQL notation
def interfaceSpec = '''Stack {
                push(Object)->Object
                pop()->Object
                peek()->Object
                size()->int}'''
study(name:'Stack-TestDrivenSelection') {
    action(name:'select', type:'Select') {
        abstraction('Stack') { // interface-driven code search
            queryForClasses interfaceSpec
            rows = 10
            excludeClassesByKeywords(['private', 'abstract'])
            excludeTestClasses()
            excludeInternalPkgs()
            // optionally, we do not want it to be a collection
            //excludeSuperClass("java.util.Collection")
            // non empty classes, i.e having complexity > 1
            filter 'complexity:[2 TO *]'
        }
    }

    action(name:'filter',type:'ArenaExecute') { // filter by tests
        sequences = [
                // parameterised sheet (SSN) with default input parameter values
                // expected values are given in first row (oracle)
                'pushPop': sheet(p1:'Stack', p2:"hello world") {
                    row  '',    'create', '?p1'
                    row '?p2',  'push',   'A1',     '?p2'
                    row '?p2',  'peek',   'A1'
                    row     1,  'size',   'A1'
                    row '?p2',  'pop',    'A1'
                    row     0,  'size',   'A1'
                }
        ]
        maxAdaptations = 1 // how many adaptations to try

        dependsOn 'select'
        includeAbstractions 'Stack'
        profile('myTdsProfile') {
            scope('class') { type = 'class' }
            environment('java8') {
                image = 'maven:3.5.4-jdk-8-alpine'
            }
        }

        whenAbstractionsReady() {
            def stack = abstractions['Stack']
            def stackSrm = srm(abstraction: stack)
            // define oracle based on expected responses in sequences
            def expectedBehaviour = toOracle(srm(abstraction: stack).sequences)
            // alternatively, use any system as a (pseudo) oracle
            def referenceImpl = toOracle(srm(abstraction: stack).systems.first())
            // returns a filtered SRM
            def matchesSrm = srm(abstraction: stack)
                    .systems // select all systems
                    .equalTo(expectedBehaviour) // functionally equivalent

            // iterate over sub-SRM
            matchesSrm.systems.each{s ->
                log("Matched class ${s.id}, ${s.packageName}.${s.name}")}
            // export to individual CSV file (if desired)
            export(matchesSrm, 'stacks.csv')
            // continue pipeline with matched systems only
            stack.systems = matchesSrm.systems
        }
    }
}
```

List. 14: LSL Script for Test-Driven Filtering of Stacks

from which Java systems are retrieved. Here the data source selected from the underlying executable corpus of the observatorium is set to Maven Central (see Section 12.4).

The global parameter definition `interfaceSpec` defines the desired interface signature of the stack abstraction as a (Groovy multi-line) string in the interface notation using LQL. Even though the interface specification can be directly defined inside the action block that uses it, it is globally defined to have the typical semantics of (global) variables (as known from general-purpose languages).

The first action defined in the study block is concerned with the interface-driven search for Java classes that match the given interface signatures of the stack abstraction. As we can see, the action *select* is instantiated on a predefined action of type `Select`. This predefined action defines the default behaviour for the creation of new abstractions (or abstraction containers) by using the `abstraction` block. In this case, the abstraction block is defined to configure an interface-driven query to return at most 10 Java classes (see Line 11 and 12 in Listing 14). As mentioned before, the action picks up the default data source configured by the script to retrieve classes from the Maven Central data source.

Optional parameters can be set in order to define custom filtering criteria, some of which are included here for demonstration purposes. For example, the returned Java classes can be filtered based on the properties that are stored in the executable corpus (Chapter 8). Once the action has been executed, a new abstraction has been created with the Java classes (i.e., candidate systems) that were returned by the interface-driven search.

Stimulus Sheets and SM Configuration

The second action depicts a predefined action that enables the execution of configured SMs in the arena. The action depends on the former action that produced the stack abstraction (i.e., data container). To configure an SM, the action block defines a (singleton) list of sequences that contains a stimulus sheet in SSN notation. The SSN notation is defined in LSL via simple LSL commands that reflect its tabular representation. Note that as well as being explicitly defined in terms of LSL commands, sequence sheets can also be loaded remotely and read in as external files (i.e., as spreadsheet documents).

Here the sequence sheet is called `pushPop` and defines two formal input parameters, the `Stack` class and the element which is pushed onto the stack. Recall that sequence sheets are invoked in the same way as classic methods from object-oriented programming languages, so both define method signatures. Inside the sequence body, input parameters are referenced via question mark value expressions (e.g.,

?p2). Note that in this particular sheet, default parameters are defined that are used to create sequence sheet invocations on Java classes. Since the sheet is parameterised, LSL allows the definition of sequence sheet variations using the following syntax —

```
1  'sheet2': sheet(name: 'pushPop', p1:'Stack', p2:5)
```

This example creates a variant of the first sheet, but instead of using the string `"hello world"` for the second parameter, `p2`, the integer value `5` is used. There are further variations of the LSL commands available that facilitate the (re)use and handling of sheets.

It is important to note that the sequence sheet also encodes the expected behaviour of the stack abstraction in terms of actuations in the first row in terms of expected output values (i.e., responses). If we assume a classic test-driven code search scenario, these are typically provided by the developer who wants to reuse a class implementation of a stack. The sequences defined in the action block are used to populate an SM based on the Java classes contained in the abstraction container. Note that the SM is manipulated via the lists of systems and sequences that may be modified within an action block.

The configuration and execution of the SM in the arena can be further customised. In this example, the local parameter `maxAdaptations` instructs the arena to only create one adaptation of a Java class (Section 9.3). Assuming that the number of Java class candidates is 10, then in this case 10 systems are populated in the SM, otherwise if the adaptation parameter is greater than 1, the resulting SM configuration contains $maxAdaptations * \#classes$ systems and one stimulus sheet.

The setting of explicit profiles instructs the arena to configure desired execution environments and/or desired scopes that determine the extent of Java classes in terms of their code elements (important for measurements that measure properties about systems). In addition to their local definition, profiles can also be defined on a global (study) scope in order to reuse them within a set of actions. Accordingly, actions can reference globally defined profiles via the unique names. Being able to define and control the execution environment and measurement scope is vital in studies that attempt to achieve replicable results (i.e., for software experimentation). In this example, the arena execution environment is configured to use a certain Java JVM version (i.e., Java 8). The actual execution environment is chosen by the "image" parameter that selects an existing container image that is then populated inside the arena (see containerisation/sandbox execution environments in Section 12.3).

The action under discussion is instantiated based on a predefined action. In this case, a custom `execute` block that preprocesses the SM is optional since the predefined action defines its own execution behaviour. As a consequence, the `ArenaExecute` action populates the SM inside the arena based on the given configuration and executes it according to the given profile in order to obtain the resulting SRM that contains all invocation records including the observations of exhibited behaviour (i.e., actuations in terms of responses).

Once the arena has executed all sequence/system pairs that are defined in the SM, the `whenAbstractionsReady` life cycle block of the action is executed (starting at Line 44 in Listing 14). LSL provides a minimal set of commands to operate on the abstraction (data) container and to manipulate the available SRMs in an imperative way using custom logic.

Firstly, abstraction containers can be accessed via a map/dictionary structure called *abstractions* that holds key-value mappings of the abstraction name (key) and container (value). These attributes are injected and set by the surrounding workflow engine and are always available to LSL developers. The resulting SRM from the arena execution can be retrieved via the `srm` command by providing a reference to the container of the abstraction. In this example, the pipeline demonstrates the test-driven selection of stack implementations based on behaviour described by a sequence sheet. The `toOracle` command can be used to create an "oracle" that represents the expected behaviour in terms of all actuations defined by some "source". In this case, the oracle is constructed from the first row of the given sequence sheet `pushPop` (in list `sequences`). Alternatively, for demonstration purposes, Line 50 in Listing 14 also shows how existing systems in the SRM can be selected as the "pseudo oracle" (i.e., as a reference implementation of a stack, see Section 3.5.3).

Note that LSL realises the SRMPATH notation introduced in Section 6.4 in terms of the "builder" pattern (GoF patterns [90]). In short, the builder pattern allows commands to be chained using the dot notation. Each invocation returns a reference to itself, so subsequent method calls to the same reference can be made. The builder pattern is also a recurring design pattern in the construction of DSLs [62]. For example, the command `srm(abstraction:stack).systems` returns the list of systems in the SRM whereas `srm(abstraction:stack).sequences` returns the list of sequences in the SRM.

The selection criteria in classic test-driven code search is to establish whether the behaviour exhibited by the classes is functionally equivalent to the behaviour expected (here encoded by the oracle that was derived from the sequence sheet).

LSL facilitates such a selection by allowing a list of systems to be compared to the expected behaviour (i.e., oracle) using the chained command `equalTo`[5].

Once the command has executed, a "filtered" SRM is returned that is limited to the matching Java classes that are functionally equivalent to the expected behaviour. The remaining statements illustrate how SRMs can be further manipulated. First, we exemplify iteration over all Java classes in the matched SRM by using Groovy's for-loop syntax `each` (i.e., using closure syntax [237]). Here we simply log attributes of the Java classes using a formatted string that is appended to a logging file that can be later retrieved by users from their "workspace".

The penultimate statement in the given script demonstrates the export of an SRM to an external CSV file that can be later loaded and processed in an external data analytics platform (Section 6.7). Finally, the last statement demonstrates how the "output" of an action that potentially flows to other actions can be manipulated. Here the abstraction container is modified to only include the systems matched by the expected behaviour. Note that the data produced by one action is immutable by design, so subsequent actions actually receive a copy of the abstraction containers that they "subscribe" to.

10.3.3 Measurements and Scopes

This section shows how the existing test-driven selection pipeline from Listing 14 can be extended to demonstrate some of the software measurements that can be conducted in the arena and stored in the SRMs. Listing 15 shows a third action *measure* that depends on the stack abstraction as well as the SRM of the previous filter action in the pipeline. It is no coincidence that it looks similar to the filter action. In fact, measurements can be directly configured in the filter action as well. However, for clarity another action that achieves the same functionality is created.

Measurements in the arena can be specified in terms of "features" in the header of the action block (i.e., the configuration of parameters of the action). Note that the configuration block is omitted. In this example, the arena is instructed to enable measurements of code coverage as well as measurements of mutation score. For code coverage, the research prototype of the observatorium integrates the popular code coverage tool library JaCoCo for Java [122], whereas PIT [153] is used to measure mutation score. Note that the integration of the aforementioned tools can be achieved in multiple ways. Here the `ArenaExecute` action acts as a compound action that integrates existing measurement harnesses. However, those harnesses can also be implemented as independent actions (i.e., analysis steps).

[5]Note that this terminology was inspired by existing unit test frameworks such as `JUnit`.

```
action(name:'measure',type:'ArenaExecute') { // measure
    features = ['cc', 'mutation'] // code coverage and mutation score

    dependsOn 'filter'
    includeAbstractions 'Stack'
    profile('myTdsProfile') {
        scope('class') {
            type = 'class'
            // ignore calls to popular logging facilities (by package name)
            pkgBlacklist = ['org.slf4j.*', 'org.apache.log4j.*', ...]
        }
        environment('java8') {
            image = 'openjdk:8-jdk-alpine'
        }
    }

    whenAbstractionsReady() {
        def stack = abstractions['Stack']
        def branchTotal = srm(abstraction: stack)
                .systems.observations['cc.branch.total']
        def mutationScores = srm(abstraction: stack)
                .systems.observations['mutation.score']

        int totalSystems = srm(abstraction: stack)
                .systems.size()
        // averages
        double branchAvg = branchTotal.mean()
        double mutationScoreAvg = mutationScores.mean()

        log("Total number of systems is ${totalSystems}")
        log("Average number of total branches is ${branchAvg}")
        log("Average mutation score is ${mutationScoreAvg}")
    }
}
```

List. 15: LSL Action Example for Analysing Software Measures

The measurement scope (Section 5.2) for the aforementioned tools is specified as part of the profile block. Here a new scope is constructed that configures the "class scope" in which only the "entry" class that defines the matching interface signatures of the stack abstraction is measured. Moreover, for the sake of illustration, any method invocations to popular logging facilities are ignored in (code coverage) measurements by maintaining a black list of known package names[6]. The environment is set up as for the previous action *filter*. Since the action does not manipulate the SM of the stack abstraction obtained from the previous action, the same SM is reused. Once the SM has been executed, the `whenAbstractionsReady` post-processing block is executed that demonstrates the analysis of the obtained measures for code coverage and mutation score constrained by the scope definition. Note that LSL only defines a minimal set of DSL commands, since sophisticated analyses are intended to be done in external data analytics platforms. Nevertheless, LSL provides convenient commands for basic analysis of measurements.

As with the filter action, a reference to the abstraction container is first obtained and then SRMPATH is used to navigate to the SRM that contains the measurement records. The final attribute "observations" is a map/dictionary structure that maps arbitrary observations by their unique name, including the obtained measurements. In this particular example, we retrieve the total number of branches for each system (via JACOCO) and the mutation scores. Next to the measurements, we also determine the total number of systems in our SRM using the SRMPATH notation (i.e., invoking "size" at the end).

Observations are always returned as single-column data frames indexed via the identifiers of the classes. Based on the data frame model, we can call a variety of summarisation/aggregation commands. In this case, we determine the average of the total number of branches and mutation scores for the number of classes (i.e., using the operation `mean`). Note that the returned measures depend on the defined scope. In this case, the default "class" scope was selected. If measures were defined using different scope definitions, LSL can be used to compare the records of two SRMs to analyse potential differences between the scopes. Accordingly, the returned measures can be compared with each other, or the summarised averages can be compared. Finally, the last three statements write the obtained information to a log file (for demonstration purposes).

[6] Ignoring method invocations to popular logging facilities can be useful at times, since it can help to make comparisons of complexity more significant (less bias), for instance.

```
1  action(name:'select', type:'Select') { ... }

2  // Nicad6 code clone detection
3  action(name: 'clones', type: 'Nicad6') {
4      cloneType = "type2"
5      collapseClones = true // remove duplicates

6      dependsOn 'select'
7      includeAbstractions 'Stack'
8  }

9  action(name:'filter',type:'ArenaExecute') {
10     dependsOn 'clones'

11     ...
12 }

13 action(name:'measure',type:'ArenaExecute') { ... }
```

List. 16: LSL Action Example for Advanced Filter Actions

10.3.4 Advanced Filter Actions

Finally, this subsection showcases the use of some advanced selection criteria in LSL pipelines to filter systems and sequences based on certain attributes. Listing 16 illustrates an advanced filter action using a predefined action that integrates the well-known code clone detection tool NICAD [54].

NICAD is a code clone detection tool that achieves high recall and high precision. It is basically able to detect code clones of type-1 to type-4 [207]. Action `clones` instantiates an LSL action based on the existing action type `Nicad6` that integrates the tool. As with any other actions, the header of the action block configures the tool to detect type-2 class clones (the source of the Java class is the subject for comparison). The second configuration parameter `collapseClones` instructs the action to remove any code clones (i.e., classes from the abstraction container), while retaining unique non-duplicate classes. If the parameter is set to false, all classes are retained in the abstraction, but a record for each class is stored that contains information about the clone classifier that NICAD assigned to each class. Two classes are regarded as clones if they have the same unique classifier (string) assigned.

The clone filtering action sits between the select and the filter action of the test-driven selection pipeline shown in Listing 14. In other words, the clone action receives the classes returned by the select action and drops any duplicates. The subsequent arena filter action then receives a filtered set of classes, since it "subscribed" to the abstraction container produced by the clone action.

This example demonstrates the "clean" design of LSL pipelines. Actions are decoupled from other actions and can be used to extend existing pipelines with additional features. In this case, we extended the classic test-driven code search

pipeline with non-trivial detection of class duplicates based on a proven (academic) code clone detection tool. As an alternative to predefined filter actions, users can provide their own plain actions that realise custom filtering strategies using LSL commands in the syntax of Java/Groovy.

10.4 Language Quality

The previous sections have demonstrated the general structure of LSL scripts used to construct analysis pipelines for the large-scale, dynamic analysis of software systems (based on the example of Java classes as systems). This section discusses LSL in terms of the classification guidelines summarised by Karsai et al. to indicate the quality of the language [134] —

- *Language Purpose*,
- *Language Realisation*,
- *Language Content*,
- *Concrete Syntax*,
- *Abstract Syntax*.

To begin with, the *purpose* of LSL as a language is tailored to the observatorium application domain. The target audience for LSL are software practitioners and researchers that want to conduct large-scale dynamic analyses of software systems. The purpose of LSL ranges from the large-scale mining of dynamic properties to derive new knowledge about big code, to sophisticated and replicable software experimentation (e.g., to support empirical software engineering). Even though LSL supports basic data analytics, it is not designed to replace sophisticated data analytics platforms that scale to large (tabular) data. Instead, LSL and the observatorium platform are designed in such a way that users can easily export SRMs of interest to their favourite data analytics platform.

The *realisation* of LSL was achieved using the dynamically typed language Groovy based on the widely-used Java programming language whose syntax it extends. As a consequence, engineers that are familiar with either of these languages can directly start writing LSL scripts with the proposed pipeline structure resembling data workflows. As well as reusing the syntax of Groovy and Java, LSL also reuses their type system (e.g., primitives and Java collections etc.). The declarative nature of actions as blocks is also similar to classic method blocks. Developers that use LSL

will have a rapid learning curve once they understand the general structure of study and action blocks.

The *content* of LSL is limited to the concepts of the observatorium application domain as described by our terminology and models. We provide a minimal set of DSL commands that is simple and expressive enough to support rich analysis pipelines of non-trivial complexity. Since we offer DSL commands based on a simple domain model of the concepts in the observatorium application domain, LSL is much more expressive than general-purpose languages. Together with the general structure of study pipelines and their reusable analysis steps (i.e., actions), LSL covers all the phases of observatorium usage, including the selection, analysis and comparison of software systems. Even though the DSL commands are concise and orthogonal, the availability of the Groovy and Java syntax makes LSL development possible in a variety of ways. For DSLs, this can be both an advantage, in terms of flexibility, but also a disadvantage, in terms of the conciseness of the language. From our experience, we would advise users of LSL to use Groovy as the main language to encode (imperative) logic. The definition of new classes or methods should be avoided wherever possible and should be put into the (Java) implementations of predefined action types (Section 12.5.2).

LSL offers a *concrete syntax* for the construction workflow in terms of creating study pipelines that contain one or more actions. The syntax and structure of LSL scripts and their containing blocks is well-defined. Actions have a well-defined structure and include a set of life cycle operations that are executed by the platform in a particular order. The structure of an action block follows typical code conventions including the initialisation of an action with a set of parameters. Since Groovy/Java syntax is available, code comments can be attached to any statement. Based on Groovy's abstract syntax tree parser, additional checks can be developed to (statically) validate LSL scripts with respect to their structure and their DAGs (validating the data flow).

Finally, LSL's *abstract syntax* is designed with modularity and extensibility in mind. As discussed before, actions are "decoupled" and can be realised independently of concrete pipeline structures. Instead of hard-wiring actions to concrete pipelines, actions subscribe for existing data and get notified when it is ready for consumption. LSL scripts are tightly integrated into the workflow engine of the observatorium. Well-defined bindings between LSL scripts and the platform allow for bidirectional communication. Actions that are defined in LSL scripts are connected with actions realised in the backend of the observatorium. The decoupling of LSL actions and their counterparts in the platform simplifies the integration of external tools and techniques. This chapter has showcased a few example tool integrations including code coverage measurement with JaCoCo, mutation score measurement with PIT as well as code clone detection with NiCad.

11 Analysing SRMs

LSL offers a dedicated pipeline definition language to define individual (dynamic) analyses and a minimal set of DSL keywords to manipulate SRMs. The script-driven analysis of SRMs, however, is designed to focus on the gathering and selection of systems and observational records and the manipulation of SRMs. Rich data analytics are therefore limited in LSL. Since state-of-the-art data analytics platforms are powerful and mature, our design philosophy is not to (re)implement data analytics in the observatorium, but to interface with the existing ecosystem of tools in the best way possible.

In this chapter, we first discuss the differences between script-driven analysis and data-driven analysis. Thereafter, we explain how SRMs represented in the common data frame format can be exploited for rich data-driven analyses of behavioural relationships. Finally, we discuss possible SRM configurations.

11.1 Creation, Execution and Analysis of SRMs

Figure 11.1 provides a high-level overview of how SRMs are processed in the observatorium, including how they are created, executed and analysed. We distinguish between the script-driven manipulation of SRMs as part of LSL analysis pipelines (see the example in the previous chapter) and the data-driven analysis of SRMs in external, data analytics platforms. While script-driven processes run inside the observatorium, the task of analysing the data is delegated to external platforms that tightly integrate with the observatorium to load SRMs of interest.

11.1.1 Script-Driven Manipulation vs Data-Driven Analysis

Since the goal of the observatorium is to support focused analyses at the scale of big code, including software experimentation, large-scale data analytics is essential. The separation of concerns (i.e., script-driven manipulation versus data-driven analysis) is a deliberate design choice based on (a) the idea that each platform does what it is best at (i.e., each platform has a different focus), and (b) the fact that separating OLTP activities from OLAP activities is an established paradigm (Chapter 6). While large-scale analytic platforms are mature and operate on versatile data

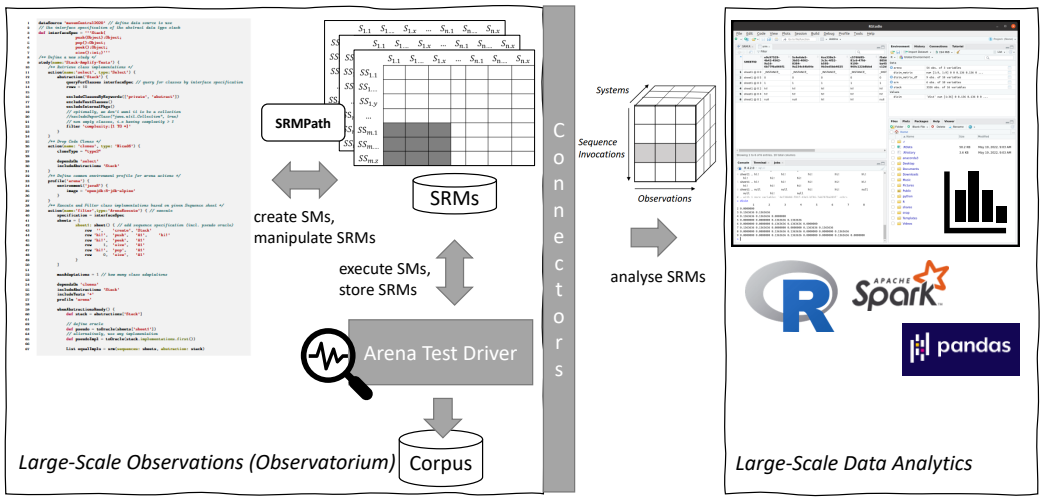


Fig. 11.1.: Creation, Execution and Analysis of SRMs

structures with a large, support ecosystem, the observatorium can focus on realising observations efficiently at a large scale.

Even though the observatorium does not (re)implement data analytics itself, its unified modelling and data representation approach provide a well-defined integration point to existing data analytics solutions. The data analytics platforms that our research prototype integrates are popular and widely used by both practitioners and researchers. By integrating these established tools users are able to work with features they are already familiar with or may opt to integrate additional platforms that support the common representation of data frames.

This approach allows users to leverage the ecosystem of data analytic tools including first-class support for rich data visualisation (e.g., plotting capabilities) that support data exploration activities as well as data illustrations to ease the interpretation of results. Regarding big code, existing platforms also have first-class support for tight integration into advanced pipelines that deploy machine learning techniques. Building those capabilities from scratch or providing custom solutions, therefore, would not only be a huge challenge but also completely unnecessary.

Apart from well-defined foci, there is another important, human-related reason why it makes sense to separate data analytics from script-driven manipulation of SRMs. Since data analytics tasks are typically creative and rely on the cognitive capabilities of humans, they cannot be fully automated. Human analysers usually have to first explore and clean large data sets in order to characterise and interpret them. Data exploration as well as the related tasks of data visualisation are activities that are done iteratively, by repeating certain commands instead of fully-elaborated

scripts. Once analysers have gained enough confidence about the underlying data, they may automate further analyses.

Since LSL pipelines are designed to be executed once, manual exploration of data cannot be included, so it has to be separated out from the process of gathering observational records at a large scale. It is important to note, however, that there are certain analysis activities that may be performed automatically. However, here we assume the current landscape of data science-related tasks which require a high level of manual effort.

At first sight, one possible way to integrate certain data analytics capabilities into the observatorium, is to integrate "ad hoc" data analytics into an analysis pipeline by calling external data analytics platforms from within the pipeline. Although this might seem convenient, however, it is not considered in the scope of this thesis. Nevertheless, LSL pipelines provide basic analytical operations such as aggregated functions that can be applied to records of SRMs which can be navigated using the SRMPATH notation available in LSL pipelines (Section 6.4).

11.2 Data-Driven Analyses

As mentioned before, the basic intention behind the script-driven manipulation of SRMs is to allow LSL pipelines to post-process SRMs to prepare them for starting data-driven analyses. Data-driven analysis, on the other hand, reads the SRM-related data from the observatorium's database, but does not alter it in that database. Of course, data-driven analysis manipulates SRMs in a variety of ways using analytical operators, but the manipulation is done on a copy of the data.

Apart from post-processing (i.e., selection/filtering of the data of interest), including data cleansing, an important capability in data analytics is to reason about multi-objective criteria. As we have discussed before, one of the most important functions of the observatorium is to estimate whether a pair of systems is functionally equivalent with respect to their exhibited behaviours. In this particular case, the notions of functional equivalence and functional similarity of systems are of particular interest. In the following subsections, we demonstrate how SRMs represented as data frames can be exploited to create (dis)similarity matrices that depict the pair-wise behavioural relationships between a set of systems based on a set of test sequences.

11.2.1 Data Frames for Interoperability

State-of-the-art data analytics tools like R [239] and APACHE SPARK [235] have matured over many years and have gained large communities that contribute to their

evolution. To enable interoperability with these, the data layer of the observatorium (see Section 6.5) supports and operates on the common data frame. A *data frame* is essentially the same thing as a relational table or a spreadsheet. It consists of columns and rows that are identified by (named) indices. A single column is a special case of a data frame where its column dimension is 1. A column is typically represented as a named vector (or list). Since our basic data structures (i.e., SRMs and sequence sheets) are two-dimensional structures of rows and columns as well, data frames are a natural way to represent them.

There are several options for loading the data obtained from the observatorium's analysis architecture and transactional database into data analytic tools. Typically, the tools offer various ways to import data from existing data sources based on an ETL-like process. The "Online Data Processing Layer" of the observatorium can be accessed by data analytics tools in two basic ways to represent SRMs as data frames and to manipulate and analyse them —

- *Common File Formats*: SRM-related data can be exported using (intermediate) file formats such as CSV (comma-separated values) files.
- *Connectors*: A variety of special connectors exist that make it possible to access the database of the observatorium in order to load data based on specific selection criteria (e.g., JDBC connections or APACHE SPARK integrations).

The mechanism used by existing tools to load data from external data sources obviously depends on the individual tool. By supporting both ways of accessing data sources identified above, the observatorium can interface with basically any of the popular tools mentioned above. Comma-separated files are a popular way to export and share data sets between tools. Their simple structure allows any tabular representation to be translated into plain text. Each cell in a row is simply separated by a separator character like a comma or a semicolon.

Since the observatorium is driven by an RDBMS to store observational transactions, external tooling can also connect to the database and load selected sets of data for efficient data-driven "offline" analyses.

11.2.2 Distance and Similarity Matrices

The matching of behaviour in unit testing tools is typically performed at run-time as part of the testing activity (i.e., test sequence execution on a system). In order to establish whether the behaviour exhibited by a pair of systems is functionally equivalent or functionally similar in a data-driven way, we need to compare the actuation records stored in SRMs. Since SRMs represent actuations as strings (Section 6.6.1), we generalise our matching problem into a string matching (comparison) problem.

String metrics is an established field of research comprising many techniques that are applied in many computer science areas, including NLP [180]. In general, string distances can be calculated based on a variety of string (distance or similarity) metrics such as metrics using edit distances (Levenshtein and Hamming distance), Q-grams (Q-gram, Cosine, Jaccard distance) or heuristic metrics (Jaro, Jaro-Winkler), some of which scale to large strings.

Distance matrices can be computed based on the concept of string distances. In our context, we can compute distance matrices from the output records stored in SRMs. Distance matrices have their roots in computer science, especially graph theory in which the main objective is to compute the pair-wise distances between the nodes of a graph. In our case, the elements of the distance matrix refer to the pairwise "distances" between the systems in an SRM computed from their actuations using a chosen string distance metric.

In this case, of course, we need to assume white box SRMs, so each record in the SRM depicts serialised output values returned by a particular method invocation of a particular sequence on a particular system. All the method outputs of one system are compared to all the method outputs of another system (based on the same set of actuations). Technically, the list (or vector) of string values in the columns of two systems in the SRM are concatenated to strings which are then compared using a string distance metric.

The resulting distance matrix $D_m = (d_{ij})$ with $1 \leq i, j \leq N$ where $N = \#R$ (the number of records for a given SRM), for a string distance metric m has the following properties —

- the matrix is symmetric (i.e., $d_{i,j} = d_{i,j}$),
- the elements on the main diagonal (top-left to bottom-right) represent self-comparisons (system $s_j \in S$ is compared with itself), hence the value 0,
- the off-diagonal elements, $d_{i,j}, i \neq j$, are greater or equal to 0 ($d_{i,j} \geq 0$).

In other words, the distance matrix is a square matrix whose distances above or below the diagonal (distances above and below are the same) are useful for identifying functionally equivalence and functionally similarity. Since the distance matrix contains the inverse values of a similarity matrix in terms of distances, we can easily transform a distance matrix into a similarity matrix by subtracting each value from 1 (i.e., $1 - D_m$). This is possible, since most of the string metrics return a normalised value in the range of $[0, 1]$.

Formally, based on the idea of "Intersection over Union" (IoU) from the Jaccard Index and its application in the field of computer vision (i.e., object detection) to

```r
library(readr) # read CSV as data frame, https://readr.tidyverse.org/
library(dplyr) # grammar for data manipulation, https://dplyr.tidyverse.org/
library(tidyr) # privoting, https://tidyr.tidyverse.org/
library(stringdist) # string distance computations

# CSV export from observatorium (alternatively, use RJDBC)
stack_records <- read_csv("stack_records.csv")

# records: from long to wide data frame format
srm <- stack_records %>% pivot_wider(names_from = SYSTEMID, values_from = VALUE)

# distance distance matrix based on Jaccard
# "dist" object (https://cran.r-project.org/web/packages/stringdist/stringdist.pdf)
distance_matrix <- stringdistmatrix(srm, method = "jaccard")
# to similarity matrix
similarity_matrix <- 1 - as.matrix(distance_matrix)
# view dendogram of distance matrix
plot(hclust(distance_matrix))

# all systems which are functionally equivalent to system `1`
as.data.frame(similarity_matrix) %>% select(`1`) %>% filter(`1` == 1.0)
# all systems which are functionally similar, but not equivalent to system `1`
as.data.frame(similarity_matrix) %>% select(`1`) %>% filter(`1` < 1.0)
# all systems which are functionally distinct to system `1`
as.data.frame(similarity_matrix) %>% select(`1`) %>% filter(`1` == 0.0)
```

List. 17: R Script to Demonstrate the Analysis of a Stack SRM

define a standard performance measure [196], we define the degree of functional similarity (FS) of two behaviours (see Section 3.4.1), g and h, as —

$$FS(A_g, A_h) = \frac{|A_g \cap A_h|}{|A_g \cup A_h|} \tag{11.1}$$

where A_g and A_h are the sets of actuations for g and h.

Accordingly, if $FS(A_g, A_h) = 0$, then g and h are said to be functionally distinct. If $0 < FS(A_g, A_h) > 1$, then the behaviour is similar to a certain degree, but it is not equivalent. If $FS(A_g, A_h) = 1$, then behaviours g and h are functionally equivalent.

11.2.3 Stack Example

In this section, we illustrate how distance and similarity matrices can be computed from SRMs using the R platform on the stack example that consists of a single stimulus sheet and 9 (Java) systems represented by their (entry) classes. The R program is provided in Listing 17 whereas the corresponding SRM that serves as its input is depicted in Table 11.1.

The SRM contains (a variant of) the sequence sheet of the stack abstraction shown in Listing 14 in Chapter 10 that invokes push, peek, pop and size methods to define a typical execution scenario of a stack. Table 11.1 shows the SRM that contains the

Tab. 11.1.: Example SRM of the Stack Abstraction based on the Stimulus Sheet in Listing 14 and 9 Java Systems

		1	2	3	4	5	6	7	8	9
1	create	_obj_	_obj_	_obj_	_obj_	_obj_	_obj_	_obj_	_obj_	_obj_
2	push	null	null	null	hi!	hi!	null	hi!	null	null
3	peek	hi!	hi!	hi!	hi!	hi!	hi!	hi!	hi!	hi!
4	size	1	1	1	1	1	1	1	1	1
5	pop	hi!	hi!	hi!	hi!	hi!	hi!	hi!	hi!	hi!
6	size	0	0	0	0	0	0	0	0	0

exhibited output (i.e., response) of the 9 systems for each method invocation of the stimulus sheet at hand.

Note that the first invocation (i.e., first row in the SRM matrix) creates a stack object. Since the identifiers of the objects that represent the instances of a class typically differ[1] (assuming that the SRM contains diverse classes), we use a special placeholder value in string serialisations to signal that a comparison should always lead to equivalence.

To begin with, the R script reads in the "raw" stack SRM as a CSV file exported from the observatorium. Since the CSV contains "raw" stack records from the (OLTP) database of the observatorium, they are represented as a data frame in the long format. To represent the records (here outputs of responses from systems) as an SRM, we first need to "pivot" them from the long format to the wide format in order to obtain the classic SRM format.

To establish whether the 9 systems are functionally equivalent, similar or distinct, we use an existing package that offers a variety of string distance metric implementations as well as the capability to create distance matrices. In this particular example, we compute the distance matrix for the SRM in Table 11.1 based on Jaccard similarity. While Table 11.2 provides the resulting distance matrix, Figure 11.2 illustrates the matrix elements in terms of a cluster hierarchy in a dendogram.

As one can see, the diagonal elements (pairwise systems) are all zero, since those represent a self-comparison of systems. We can now either look below or above the diagonal to spot the "string distance" between each pair of systems that was computed based on their exhibited outputs (i.e., each column of the SRM depicts the list of outputs that are compared to other lists of outputs).

The dendogram illustrates a tree of the arrangement of clusters in the distance matrix. In our particular example, there are two clusters of functionally equivalent systems. However, the two clusters are also highly similar (i.e., 86%) to each other (i.e., they have a 14% disagreement on all the outputs). This is because there is no

[1]The Java language, for instance, uses a generated hash code representation.

Tab. 11.2.: Distance Matrix of SRM in Table 11.1 Based on Jaccard Similarity

System	1	2	3	4	5	6	7	8	9
1	**0.00**	0.00	0.00	0.14	0.14	0.00	0.14	0.00	0.00
2	0.00	**0.00**	0.00	0.14	0.14	0.00	0.14	0.00	0.00
3	0.00	0.00	**0.00**	0.14	0.14	0.00	0.14	0.00	0.00
4	0.14	0.14	0.14	**0.00**	0.00	0.14	0.00	0.14	0.14
5	0.14	0.14	0.14	0.00	**0.00**	0.14	0.00	0.14	0.14
6	0.00	0.00	0.00	0.14	0.14	**0.00**	0.14	0.00	0.00
7	0.14	0.14	0.14	0.00	0.00	0.14	**0.00**	0.14	0.14
8	0.00	0.00	0.00	0.14	0.14	0.00	0.14	**0.00**	0.00
9	0.00	0.00	0.00	0.14	0.14	0.00	0.14	0.00	**0.00**

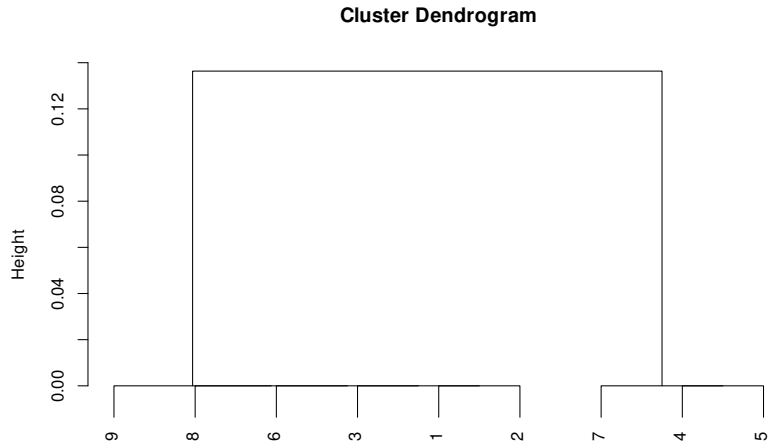


Fig. 11.2.: Dendogram of Distance Matrix in Table 11.2

functionally distinct system in our example of 9 systems. We can easily compute the similarity matrix from the distance matrix (i.e., that contains the inverse of similarity) given in Table 11.3 by subtracting the distance matrix from 1.

Tab. 11.3.: Similarity Matrix Derived from the Distance Matrix in Table 11.2

System	1	2	3	4	5	6	7	8	9
1	**1.00**	1.00	1.00	0.86	0.86	1.00	0.86	1.00	1.00
2	1.00	**1.00**	1.00	0.86	0.86	1.00	0.86	1.00	1.00
3	1.00	1.00	**1.00**	0.86	0.86	1.00	0.86	1.00	1.00
4	0.86	0.86	0.86	**1.00**	1.00	0.86	1.00	0.86	0.86
5	0.86	0.86	0.86	1.00	**1.00**	0.86	1.00	0.86	0.86
6	1.00	1.00	1.00	0.86	0.86	**1.00**	0.86	1.00	1.00
7	0.86	0.86	0.86	1.00	1.00	0.86	**1.00**	0.86	0.86
8	1.00	1.00	1.00	0.86	0.86	1.00	0.86	**1.00**	1.00
9	1.00	1.00	1.00	0.86	0.86	1.00	0.86	1.00	**1.00**

The similarity matrix then depicts indicator measures about the pairwise behavioural relationships between the systems. If the similarity measure equals 1, then a pair of systems is functionally equivalent with respect to the actuations at hand. If the similarity measure is between 0 and 1, then a pair of systems is functionally similar to a certain degree, otherwise if the similarity measure is 0, they are functionally distinct (i.e., no outputs are matched).

The similarity measures can be used to select matching relationships between certain systems as illustrated by the final three commands in Listing 17. Here we show the selection of systems that are functionally equivalent, similar or distinct to the given system identified by "1". More generally, the assumption underlying such a comparison is that system "1" depicts the "oracle" of the functional abstraction at hand, represented by a reference implementation.

Distance matrices can be computed from any number of SRMs that contain the same set of systems and the same (sub)set of actuations. In this case, SRMs and their data frames can be "joined" by simply appending all rows to a single large SRM that is then subject to a distance matrix computation. Likewise, if human analysers are only interested in the output of certain sequence sheets or certain method invocations within them, SRMs can be filtered using the slicing and dicing operations (cf. analytical cube operations in Section 6.7.1). One potential form of method invocation that is typically not of interest in comparisons is the output of the special `create` method, since it simply holds equivalent placeholder values (i.e., to signal that an object was created).

Note that navigation around SRM cubes in data frames is possible using the unique identifier (columns and rows) for systems, sequence sheets and method invocations

etc. The data contained in the exported CSV file contains those unique identifiers in a "flattened" representation. Certain method invocation records are identified though the construction of compound identifiers (keys). Based on the identifiers, several data frames can be "joined" via rows (adding more sequence and method invocations) or columns (adding more systems), thereby allowing many SRMs to be combined into one.

11.2.4 Multi-Objective Analyses

SRMs may contain records (i.e., analysis attributes) other than those related to behaviour (i.e., actuations). The observatorium allows rich selection criteria to be formulated that pursue one or more objectives. Let us assume that we have identified functionally equivalent systems based on the similarity matrix approach discussed above. Further, assume that one of the systems is considered to be the reference implementation for the stack functional abstraction at hand (i.e., serves as the executable specification of it).

In order to determine whether those systems are also implementationally distinct (Section 14.1), we may consider a single indicator measure such as the number of branches (e.g., determined through scope-aware measurements). We can select the corresponding records of the SRM, otherwise if we consider multiple measures we may look at those "independently" or we may need a way to aggregate them.

In the presence of multiple selection criteria, users can choose from a wide range of possible techniques including weighted-sum (i.e., assign a weight to each measure), or classification techniques in terms of multi-criteria clustering [168] (e.g., k-Means).

11.2.5 Labelled Data - Grouping

Some analyses performed in the observatorium produce data that is neither an output value of a response nor a numerical metric measure, but some sort of classified data often based on the assignment of labels. Code clone detectors like NICAD [54], for example, group code duplicates based on labelled classes. More abstractly, source code hashing also indirectly labels code units such as class bodies and method bodies using a generated hash sum (e.g., represented as a string value).

In either case, labels are assigned to each system in the SRM which can be compared for equivalence (string or numerical). Using labels, a straightforward approach to identify the number of clones of a given system in a data-driven way is to group systems by their label, then to summarise the groups based on the number of systems in each group. Clone systems can be filtered out by picking a representative

of each clone group and dropping all the remaining clone systems. Likewise, certain labels can be used to select certain groups of interest.

11.3 SRM Configurations

The actual configuration of an SM in terms of sequence sheets and systems depends on the analysis goal. In general, we can characterise the space of possible configurations in a variety of ways. Most fundamentally, SMs differ based on the number of columns (i.e., systems) and the number of rows (i.e., stimulation sheets). SM configurations that differ by the *number of systems* can be classified into two categories —

- *Single-System SMs*: Classic software testing activities involving a single system of interest,
- *Many-System SMs*: Establishing behavioural relationships such as functional equivalence and similarity between a (sub)set of systems.

The former category covers software testing activities that involve a single system including classic unit testing, code coverage measurements (see motivational example in Section 2.1) as well as test generation. The latter category includes analyses such as the test-driven selection of systems, classic mutation testing or (semantics-preserving) code refactorings, all of which operate on the true behaviour exhibited by software systems. While a many-system SM for test-driven selection typically contains a set of alternative systems harvested from the executable corpus, the SM for mutation testing contains the mutant systems generated from a single system of interest.

The purpose of an analysis task may be further specified. Test-driven selection, for instance, may either assume a functional abstraction of interest or may attempt to identify groups of implementationally distinct systems based on certain criteria.

Many analyses in the observatorium take advantage of test-driven selection. As test-driven selection based on a functional abstraction of interest using interface-driven code search typically requires the adaptation of systems (Section 9.3), SMs can be characterised based on whether they contain adapted systems or not. Since many possible adapted systems can be derived from a non-trivial interface, *adapted* SM configurations typically define a large set of systems that can be limited by defining suitable thresholds (e.g., capping the number of adapted systems of a single system by setting a threshold).

Another way to characterise SM configurations is to distinguish between *intra-SRM analyses and inter-SRM analyses*. While the former analyse the systems in a

single SRM, the latter analyse systems from multiple "compatible" SRMs (assuming that the systems and types of records match). An example of intra-SRM analysis is test-driven selection, while software experimentation and the comparison of two tools based on some evaluation criteria is an example of an inter-SRM analysis.

11.3.1 Manipulation

Apart from their basic characterisation, new SRMs can be produced by combining the rows of one or more SRMs (assuming that the systems match). In this case, the "super" SRM contains more sequence- and method invocations. As well as adding more rows, users of the observatorium may be interested in merging existing SRMs. In this regard, SRMs may be extended by adding more dimensions (i.e., facts in the SRM cube).

Instead of comparing the elements of SRMs directly, two or more SRMs may be summarised first, and their results compared afterwards. AUTG tools, for instance, may be compared based on their average code coverage measures. Assuming that we create an SRM for each AUTG tool, the code coverage measurements in the SRM can be summarised to compute their mean (e.g., mean branch coverage). Thereafter, the two means may be compared to assess the efficiency of each AUTG with respect to test quality (software experimentation).

As explained in the previous chapter about analysis pipelines, SMs and SRMs are often processed in a consecutive manner. An SRM produced as the output of one analysis step may serve as the input for a consecutive analysis step. SRMs can therefore be thought of as evolving along the analysis pipeline. An example of this are code-driven searches (Section 9.2) in which the input to a test-driven selection is a reference system that serves as the functional abstraction of interest. For the reference system, a set of test sequences is generated using AUTG in order to describe its behaviour. The resulting SRM serves as the input for the consecutive test-driven selection step where behavioural relationships to other systems are established.

11.3.2 Local Analysis versus Global Mining

Up to this point, we have assumed a "local" scope for focused analyses in which SMs are configured and the resulting SRMs are analysed. Such "local" mining of data is not the only way of exploiting the records of SRMs. The well-defined structure of SRMs lend themselves to more powerful analyses at the "global" scope of mining activities.

There is, therefore, a space of analyses with huge potential that may benefit from analysing the "big picture" in terms of conducting mass-analysis of SRMs at

a “global” mining level. Since SRMs contain domain-specific information, it can be exploited at a large scale to develop “big code” techniques such as new approaches for improving AUTG tools (mining stimuli from sequence sheets) or to create domain-specific oracles (mining actuations from sequence sheets) [151]. Moreover, these analysis approaches are not limited to the development of new techniques or tools. Researchers can also use them to answer research questions about properties of repositories (similar to BOA [75]).

Part V

Demonstration and Evaluation

12 Prototype Platform

This chapter presents the prototype platform, LASSO (*La*rge-*S*cale *S*oftware *O*bservatorium), developed as part of this thesis research to demonstrate the practical feasibility of the envisaged observatorium. The prototype implements the models and approaches described in the previous chapters, and therefore demonstrates the validity of Hypothesis 1.

To guide the presentation of LASSO, we first introduce its high-level, distributed architecture by explaining the core components of its four layers. Thereafter, we discuss how scalability is achieved, followed by a description of how it provides a secure and controlled execution environment in the arena. Then we introduce the prototype's executable corpus that facilitates the selection of executable Java systems (i.e., classes). The chapter proceeds with a description of LASSO's Action API for the development of LSL actions, and the realisation of the sequence execution engine that executes SMs and produces SRMs. Finally, the chapter concludes with a discussion of the extensibility options offered by the platform.

12.1 Architecture

Figure 12.1 presents a high-level overview of LASSO's controller/worker[1] layered architecture. The four layers and their components were derived from the requirements of the high-level, distributed analysis architecture introduced in Section 6.5, and from the approaches described in the previous chapters. Technically, the presented architecture contains several sublayers such as the executable corpus, build automation, job and task scheduling and collector framework for measurements etc. For the sake of simplicity, we represent those sublayers as components of four basic layers —

1. *Workflow Engine Layer*: The management layer of the observatorium which executes LSL pipeline scripts, schedules jobs and tasks, manages the executable corpus, integrates tools/techniques in terms of actions as well as manages the sandbox environments based on containerisation for the controlled execution of SMs.

[1] Historically, this was referred to as the "master/slave" model.

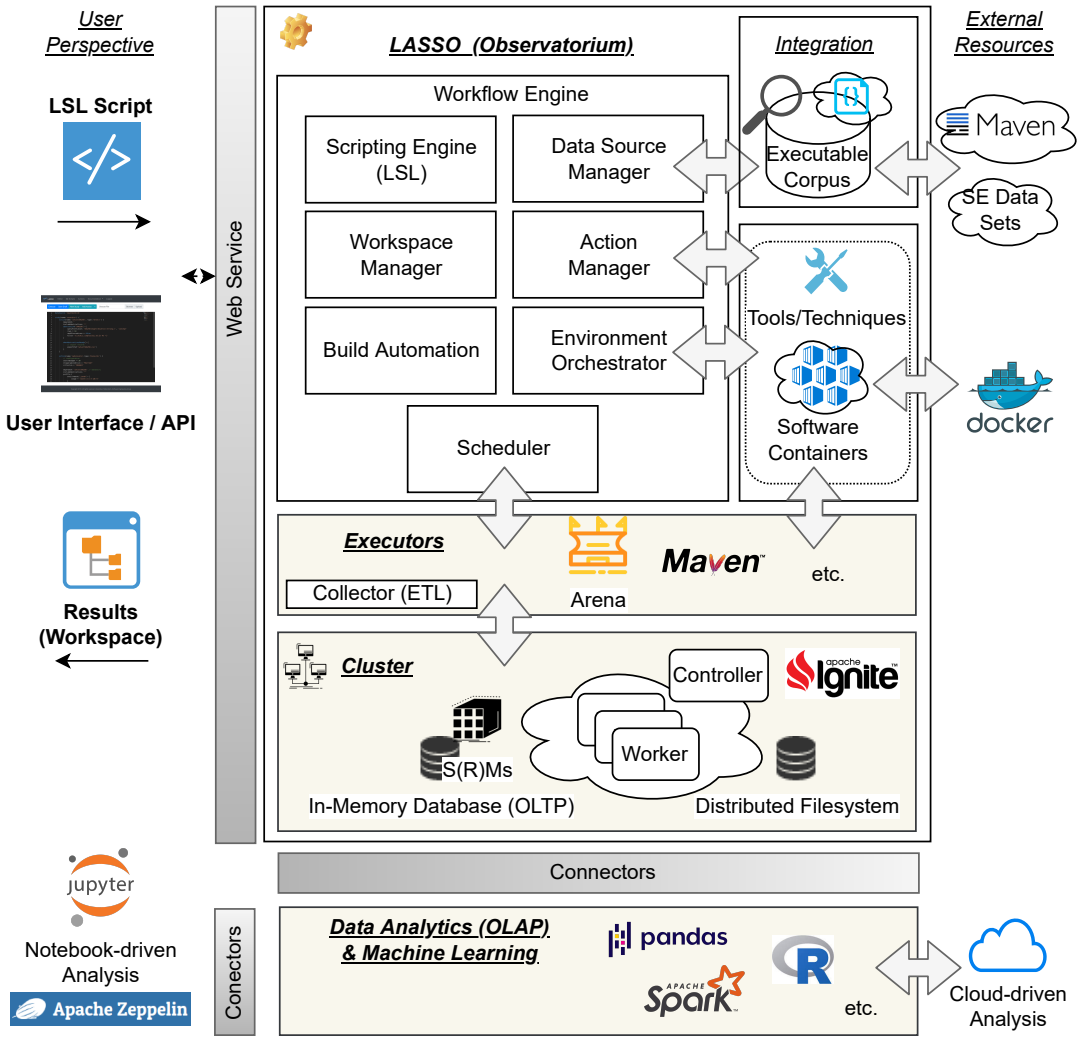


Fig. 12.1.: LASSO's Distributed Architecture - An Overview

2. *Execution Layer*: The engine which executes SMs defined in LSL scripts to produce the resulting SRMs.

3. *Cluster Middleware and Storage Layer*: The cluster middleware and storage framework which provides a distributed network of machines operating on an in-memory database and filesystem to store SRM execution data.

4. *Data Analytics Layer*: The large-scale data analytics (OLAP) service provided through external platforms to analyse SRMs using the efficient representation of data frames.

The prototype is realised as a *RESTful* web service in Java based on the Spring framework [245]. Spring enables LASSO to automatically configure its architecture of components based on the principle of IoC. At the time of writing, the prototype supports software systems written in the Java programming language. The executable corpus of LASSO, therefore, is populated with Java classes by incorporating (1) Java-specific source code artefacts from Maven Central [217], and (2) additional software engineering data sets frequently used in software engineering studies.

12.1.1 Usage

Script authors submit their LSL scripts through LASSO's web service using either its web application or its web-based Java client. For each script execution, a remote "workspace" environment is initialised to keep track of all resources and results created as part of the script's workflow execution (i.e., analysis steps defined as actions). The workspace can be accessed by a LASSO user once the script has been executed. Its contents can be browsed and examined remotely and downloaded to the user's computer. Moreover, data analytics platforms can connect to LASSO to retrieve SRM-related data via standardised interfaces (i.e., connectors) such as *JDBC* connections using a concrete *JDBC* driver that is offered by LASSO's cluster middleware. The data analytics layer itself also provides a set of connectors that tightly integrates with popular notebook-driven data science solutions supporting reproducible interactive data analytics, such as JUPYTER [146] or APACHE ZEPPELIN [236].

Each of the four layers and their contained components are discussed in greater detail in the following subsections.

12.1.2 Workflow Engine Layer

The workflow engine is the "glue" that holds the different parts of LASSO together. It manages the script's workflow and execution, and delegates work to the other

components. The *scripting engine* is responsible for the evaluation and execution of LSL scripts (Chapter 10), whereas the *workspace manager* provisions workspaces for script executions. The *data source manager* connects to the executable corpus (Chapter 7) of LASSO. It is the integration point for new data sources (i.e., code providers) and manages the data sources configured in LSL script executions. For this, it provides a query layer in order to conduct text-based queries as described in Chapter 8.

The *action manager* provides a registry of available "predefined" (reusable) actions that are available in pipelines and manages their state. Actions managed by the action manager refer to the backend counterpart of actions which are instantiated by LSL actions within LSL pipelines. "Backend" actions are designed to integrate with (external) tools and (research) techniques, so they can be "mapped" and used inside LSL scripts to perform specific analyses and comparisons of systems. Accordingly, this separation allows LSL authors and integrators to develop and share custom actions for their custom tools/techniques.

The component for *build automation* automatically generates Maven projects from selected Java systems based on build script synthesis as described in Section 7.4 for the execution layer. Sandbox environments for the execution of SMs in the arena are managed and provisioned by the *environment orchestrator*. It builds on the idea of containerisation in order to initialise controlled and secure execution environments that are defined via environmental profiles within LSL actions.

Finally, the *scheduler* component attempts to create optimal execution plans for LSL scripts based on the availability of computing resources. To this end, it uses a set of execution strategies that aim to optimise the scaling of workloads with respect to the scheduling of LSL script executions, and the distribution of tasks to cluster nodes in order to scale (a) the execution of pipeline actions, and (b) the execution of systems in the arena.

12.1.3 Execution Layer

As its name implies, the execution layer is concerned with everything related to the execution of SMs in the arena that are created by LSL actions. More specifically, based on the (Maven) projects generated by the build automation component, this layer attempts to make all systems configured in SMs executable. LASSO provides two basic types of test drivers that realise the execution of SMs in the arena —

- *Arena Test Driver*,
- *Native Test Drivers*.

The former test driver is the default executor of LASSO that takes an SM and executes it in order to create an SRM. In addition to its primary input of SMs, the arena test driver receives additional configuration parameters such as information about defined measurement scopes, environmental settings as well as specific configuration parameters regarding adaptation (see setup of pipeline and actions in Section 10.3).

As explained in Section 10.3, based on a given scope definition, the arena executor also offers an integrated measurement facility to determine the boundary of software systems once it has been executed. Based on this capability, the test driver offers various extension points to integrate (external) measurement harnesses.

Some external tools and techniques make it necessary to run the systems and/or sequences defined by SMs using a custom, so-called "native" test driver (or executor). In this situation, these come with their own executables. For example, the NICAD code clone detector action requires LASSO to evaluate the source code of Java classes using the NICAD tool. In this case, the tool is executed on a set of Java classes to detect code duplicates. Such "native" executors can be easily integrated into the platform by deploying them in a custom execution container.

Another example of a native executor is the Maven executor. Many existing tools are available as Maven plug-ins and can be integrated as such. Since our build automation technology is based on Maven, plug-ins can be simply defined and executed using Maven to collect records for SRMs.

Finally, LASSO employs a collector framework that materialises the ETL collector approach described in Section 6.5. The "raw" records obtained by the test drivers are collected centrally and passed to post-processors that create structured records for storage in the SRM representation. The ETL process including the transformation of raw records can be customised on a case-by-case basis in actions. Based on the workflow engine and its event-driven architecture, actions register for certain "raw records" and get notified once they have been collected and are available for processing. Actions can then define their own strategies to transform the records (i.e., execution data) into a structured format that is then stored in SRMs (in the storage layer).

12.1.4 Cluster Middleware Layer

LASSO's distributed architecture is powered by APACHE IGNITE [232] which is a full-blown clustering technology supporting vertical and horizontal scaling of workloads. Since LASSO is based on a controller/worker architecture, the controller node in LASSO acts as a "load balancer" which in this case orchestrates the workload of actions defined by an LSL pipeline among a set of worker nodes in the cluster.

Each node, including controller/worker nodes, use communication channels to (automatically) report their availability (discovery), computing capabilities (i.e., available resources), health and their current execution state.

The nodes use IGNITE's distributed, in-memory data grid and file system to share and store SRM-related records and (raw) execution traces collected over the LSL pipelines. Here, the distributed in-memory data grid represents the OLTP database that processes observational transactions (Section 6.6). The cluster middleware provides a query layer that supports (1) SQL-like queries, and (2) a general-purpose, distributed key-value store. The state of the grid is persisted at any time over the number of nodes that participate in the cluster, allowing the database to scale with the total number of nodes available.

12.1.5 Data Analytics (OLAP) Layer

Recall that the LASSO platform is "decoupled" from the data analytics layer (Chapter 11). Large-scale data analytics are made possible via the integration of external platforms that interface with the LASSO platform. From an end-user perspective, SRM-related data stored in the in-memory data grid of the LASSO platform can be obtained in various ways, including —

- running, "ad hoc" SQL query statements (based on OLTP schema, see Section 6.6),
- downloading CSV exports from the script's (remote) workspace that were defined in LSL scripts,
- connecting external analytic platforms directly to the in-memory grid using JDBC or IGNITE platform integrations.

IGNITE's cluster middleware provides a set of useful integrations to interface with popular analytics platforms. Since IGNITE itself is based on Java technology, its in-memory data grid can be accessed using a classic JDBC connection. For this purpose, IGNITE offers a custom JDBC driver that can be loaded from external tools in order to query SRM-related data from script executions. Even data analytics tools such as R that do not support the Java language can connect via the JDBC protocol. Moreover, IGNITE also offers first-class integration of APACHE SPARK, a popular large-scale data analytics platform that provides rich manipulation of data frames and a rich ecosystem of algorithms and techniques to drive machine learning pipelines. Even though SPARK has its own analytics DSL based on the Scala language, which in turn is based on the Java language, there are bindings available for other popular languages as well, such as Python (e.g., via PANDAS) and R.

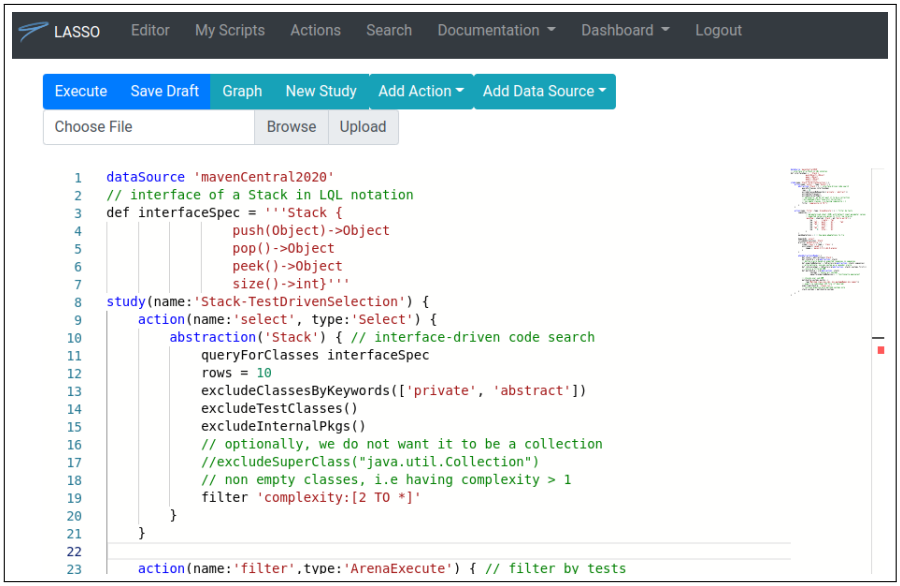


Fig. 12.2.: LASSO's Web Frontend

External data platforms offer a variety of scaling options. For instance, cloud integrations can be used to scale the demand of certain data analyses. In addition to scalability, these external tools and languages also integrate well with interactive, notebook-driven analysis tools such as JUPYTER. Notebooks can be shared as "living documents" in order to replicate analytics results (e.g., the curation of data sets or the replication of studies) [146].

12.1.6 User Interface

For end-users, there are two basic ways to interact with LASSO. As illustrated in Figure 12.2, LASSO provides a web-based HTML5 frontend which comes with an LSL script editor supporting syntax highlighting, code recommendations and the viewing of constructed pipelines in terms of dependency DAGs (see Section 10.2.1). The frontend allows users to submit LSL scripts, track their status and download SRMs and other results.

Alternatively, since the frontend is simply a consumer of LASSO's web-service API, LSL studies can also be submitted and read using any language-specific client stub (currently, a Java client stub is available). An example is the search service LASSO SEARCH that provides a web-based frontend as well as an integration of LASSO into an IDE (see Chapter 13).

12.2 Scalability

The execution of LSL scripts is job-driven based on the principle of batch processing. Batch processing is the key enabler to run the analysis steps defined in the LSL pipeline in a distributed manner in order to attain scalability. Once an LSL script has been submitted to LASSO, the scheduler of the controller node first creates a job description and puts the "job" in a waiting queue. The job is picked up automatically based on a particular *job scheduling policy* that basically checks whether the current cluster workload is high or low.

12.2.1 Execution Plans and Load Balancing

Apart from the scheduling of jobs, the controller node also evaluates LSL scripts in order to validate them and derive their dependency DAGs. These steps are necessary to create a distributed execution plan in which actions and systems are assigned to a set of available worker nodes in the cluster. As part of the execution plan, each action is assigned a *partitioning strategy* that defines how blocks (i.e., sets of systems defined by SMs) are allocated to worker nodes.

The actual LSL script execution is performed centrally on the controller node, whereas actions defined by it are executed on allocated worker nodes. Each worker node then executes the entire action on one "block" of systems (i.e., execution layer). The controller node ensures consistency by joining the results and state after all worker nodes have completed their actions (state is shared through an in-memory data grid and distributed file system). The default partitioning strategy creates "blocks" of systems in a round-robin fashion which are then allocated to worker nodes. Partitioning strategies, however, can be further customised as explained below.

Partitioning Strategies

The allocation of worker nodes depends on the partitioning strategy of a certain action. Actions can define their own partitioning strategy based on their particular needs. There are three core attributes that can be modified in order to influence the partitioning of systems and their allocation to cluster nodes —

- *Controller vs Worker Nodes*: Actions can be executed either "locally"[2] on the controller node or "remotely" on a worker node.
- *Enable/Disable Partitioning*: Partitioning may be disabled if undesired (or incompatible) for certain analyses.

[2] "Locally" refers to the execution context of the LSL script that is executed on the controller node.

- *Partition Schema*: The actual schema applied in order to allocate blocks of systems to a set of worker nodes.

The first attribute decides where (i.e., at which "location") an action is actually executed. Some actions do not need to be executed in remote worker nodes because it is more efficient to execute them directly as part of the LSL script execution on the controller node. These special actions include the `Select` action that retrieves systems from the executable corpus. Since the retrieval step is an atomic operation from the perspective of a "client" that receives the result, there is no need to distribute this action. The same situation applies to "plain" LSL actions that are not instantiated at a predefined action time. These actions usually serve as intermediate analysis steps that manipulate SRMs.

The second attribute of a partitioning strategy decides whether or not a partitioning schema is applied. Note that not all types of analysis are "distributable" per se. One example is the integration of the NICAD code clone detection tool that requires the source code of all software systems to be in one place. In general, the unpartitionability of systems does not harm scalability since tool processes (cf. native test drivers) can be run in parallel on a set of remote nodes. These kinds of analyses can basically be scaled as well, therefore, but not "by design".

The third attribute assumes that the workload of an action can be distributed so that blocks of systems can be allocated to a set of worker nodes. It defines the actual schema that is applied in order to create a partition of systems. This presents the default strategy for the execution of sequence/system pairs of SMs.

Many partitioning schemes are possible considering the number of functional abstractions and SMs involved. If an action handles more than one functional abstraction, a partition based on the number of functional abstractions or stimulus matrices can be created. Alternatively, one may partition based on either the columns (i.e., systems) or the rows (i.e., sequence sheets) of an SM. An important determinant of a partition schema is the size of the blocks. LASSO considers two basic options in this case. The first option creates blocks of equal size, where size is determined by the number of "free" worker nodes. The second option, employs a typical resource-based partitioning approach that is also used by general-purpose computing grids [72]. Here the block size for each "free" worker node is determined based on its computing power. To this end, each worker node shares a table of its available computing resources in terms of CPU cores and memory. Similarly, the available computing resources can also be constrained for each script execution (i.e., limited), or may be used to define a prioritisation schema that can "rank" available worker nodes by their available resources and computing power. The default partitioning scheme used by LASSO creates "blocks" of systems of equal size based on the number of

"free" worker nodes in a round-robin fashion which are then allocated to worker nodes.

12.2.2 Build Automation at Scale

LASSO supports scalable, automated build automation and build script synthesis by using Maven 3 as the build tool ecosystem (Section 7.4). This has the advantage that LASSO can leverage Maven's rich dependency management mechanism, plug-in ecosystem (including a variety of analysis plug-ins etc.), and established systematic reporting system. Generally, for each Java system retrieved, LASSO synthesises a project build script by generating a Maven compatible project object model (i.e., `pom.xml`). The synthesis of the build scripts can be modified and controlled by each individual LSL action. In order to manage parallel builds on each worker node, LASSO extends Maven with a custom "event spy" implementation which is able to provide event-based reporting of builds currently run by Maven. By default, each worker node is able to run many builds concurrently (multi-threading) to improve efficacy. Since systems and their execution of builds are allocated to a cluster of worker nodes, Maven builds not only run in parallel efficiently, but also scale horizontally.

As discussed previously, in the execution layer many actions that require "native test drivers" may be implemented using Maven's rich ecosystem of plug-ins. For example, LASSO supports "classic" measurement of test coverage criteria and traditional unit testing by relying on readily available Maven plug-ins. Similarly, the AUTG tool EvoSuite is also integrated via its existing Maven plug-in.

12.3 Sandbox Execution Environments

The secure sandbox environment for executing actions and systems is based on DOCKER containerisation at the operating system level. It supports the specification of controllable (e.g., operating system, Java version etc.), reusable and isolated run-time environments and enables fine-grained control over resource allocation and permissions [39]. The advantage of DOCKER is that execution environments can be easily constructed and shared as downloadable container "images" through DOCKER repositories (e.g., DOCKER HUB), or alternatively, created through custom container configurations. This facilitates the integration of new execution environments and new analysis tools into LASSO on-the-fly.

To integrate custom containers, the LASSO platform runs its own private docker repository from which preconfigured containers can be fetched. LSL users can then specify these in their LSL script profiles to specify the execution environments. The

use of containers ensures that subsets of systems that are allocated to different worker nodes (i.e., remote machines) run in the same execution environment using the same set of constraints[3].

12.4 Executable Corpus (Data Sources)

Based on the approach described in Chapter 7, LASSO's executable corpus is split in two data-related parts —

- a queryable SOLR/LUCENE index of harvested Java artefacts (i.e., classes and its methods) including their interesting analysis attributes,
- an artefact repository that stores executable artefacts in the way explained in Chapter 7 by taking advantage of the Maven repository model.

The former serves as the database that is queried by LSL scripts (e.g., enables IDCS etc.), whereas the latter contains all the (executable) artefacts harvested from external repositories in a format that is understood by the platform.

The corpus is created using three components that resemble the ETL process, (1) a crawler that downloads artefacts of interest from repositories, (2) an analyser component that inspects and analyses the artefacts, and (3) an index component that translates the analysis results into a structured representation that is stored in the queryable index structure.

LASSO's underlying executable corpus contains a set of artefacts that were harvested from Maven Central and other software engineering corpora. As discussed for LSL pipelines, developers of LSL scripts can choose the "data sources" of interest to them. Using the corresponding LSL command, one or more specific data sources can be selected from which Java classes or methods are returned via text-based queries.

12.4.1 Maven Central

As its name implies, the Maven Central data source contains Java software artefacts harvested from the Maven Central repository [217]. This artefact repository contains a large range of popular third party Open Source libraries including Spring, Apache Commons and Google Guava etc. which are used and published by Open Source as well as industry practitioners. Packaged Maven artefacts can contain a variety of types such as binary code artefacts (`*.class`), plain source code artifacts (`*.java`) as well as supplementary artefacts such as textual documentation containing rich metadata [195]. One of the most important data points are dependent artefacts.

[3]Obviously, the available computing power needs to be controlled as well.

Tab. 12.1.: LASSO's Maven Central Corpus Statistics

Unit	Total	Unique
Artefacts	184,464	184,464
Compilation Units	8,884,430	6,947,672
Classes (non-abstract)	6,682,724	5,281,170
Classes (abstract)	531,732	397,691
Constructors	10,700,527	4,064,347
Methods (non-abstract)	75,335,199	28,916,079

Typically, the repository contains several releases of an artefact [194]. As part of the ETL process, LASSO's Maven Central crawler harvested Java code from the most recent Maven artefacts that contain plain source code (i.e., consisting of `*.java` files) as well as byte code artefacts (i.e., consisting of `*.class`). LASSO's analyser conducted extensive analysis of the code components. By design, the executable corpus and Maven Central use the same Maven repository model. So in this particular case it is not necessary to (re)package artefacts. The artefacts of Maven Central are simply copied over into the artefact repository of the executable corpus.

The key statistics for LASSO's Maven Central data source are shown in Table 12.1. It contains $184,464$ indexed Maven artefacts, $8,884,430$ indexed Java compilation units (i.e., stored as class documents) and $75,335,199$ indexed Java (non-abstract) methods (i.e., stored as method documents).

It is important to note that the number of "unique" class-/method bodies is considerably lower once code duplicates have been rejected through simple string hash comparison. In large repositories like Maven Central, identical code clones often arise due to copy/paste reuse [212], multiple releases of related artefacts (i.e., software projects and their modules that share a certain code base), or project forks promoted by social coding activities.

12.4.2 Software Engineering Corpora

The benchmarking of tools and techniques to support experimentation (Chapter 15) require the integration of existing (manually) curated corpora used in previous studies. Their integration into the executable corpus offer three opportunities. Firstly, existing studies can be translated into LSL pipelines in order to replicate the results of existing studies. Secondly, new tools and techniques can be studied using LSL pipelines that use these data sets. Thirdly, when performing experiments using the analysis capabilities of the observatorium, existing curated data sets themselves

Tab. 12.2.: Software Engineering Corpora

Corpus	#Projects	#Indexed	Remarks
50K-C [172]	50,000	49,785	
Defects4J [133]	17	17 (+ versions)	was already available in target repository format
NJR [185] (1.0)	293	293	authors envision 100K projects, but prototype contained 293 projects only
Ohloh [179]	49	19	
Qualitas.class [231]	111	31	compiled version of Qualitas [229]
SF110 [83]	110	110	projects structured to Maven conventions
SIR [71]	68	68	C++ and PHP projects were ignored
Sourcerer [162]	19,173	7,093	highly heterogeneous project structure

can be further explored and characterised based on their "inherent", yet unknown, properties like their level of diversity [179].

Table 12.2 presents an overview of the software engineering corpora selected for integration into LASSO based on the following criteria —

- *Relevance*: Relevance to experimentation in software engineering. Relevance was determined based on the number of "usages" in terms of literature citations.
- *Availability*: Public availability (i.e., there should be no restrictions on usage).
- *Programming Language*: Provision of Java code. Since our research prototype currently supports Java only, only data sets that contain Java source code are relevant.
- *Completeness*: Availability of some kind of processable project structures in order to produce executable artefacts (i.e., satisfy compilability). Plain source code must also be available.

Note that other popular corpora such as those used for the evaluation of code clone techniques (e.g., [226]) have not been included since they contain (incomplete) systems (e.g., code snippets or single Java classes where the project-related context is missing).

Data sets are integrated using a dedicated analysis and transformation pipeline based on the ETL process (i.e., "mavenisation" process) described in Section 7.3. The pipeline aims to automatically preprocess the raw artefacts in the selected, heterogeneous data sets. Maven artefacts produced by this pipeline include the compiled code, plain source code and data set specific metadata. These are then deployed into LASSO's artefact repository which LASSO's internal ETL pipeline can index (same as for the indexation of Maven Central).

The high success rates presented in Table 12.2 suggest the basic feasibility of LASSO's executable corpus creation approach. More specifically, it demonstrates that data sets (i.e., corpora) with heterogeneous repository and project layouts can indeed be integrated into a single, executable corpus. Once a data set has been transformed and deployed into the common (Maven) repository model, the standardised ETL-based corpus creation process can be applied in order to index the deployed artefacts to enable their text-based retrieval.

12.5 Actions

LSL pipelines consist of a set of actions (i.e., analysis steps) that are chained together to facilitate focused analyses. In this section, we first explain how the Actions API of the LASSO prototype works, and then we provide a summary of all the actions that are available in the prototype implementation.

12.5.1 Actions API

Each LSL action defined in an LSL study script has a corresponding Java class counterpart in the "backend" of LASSO. Each Java class counterpart supports an existing API that we refer to as the "Actions API". Since the platform operates on the principle of IoC, this API allows the platform to manage actions in a uniform way using a well-defined life cycle of actions, from their creation to their destruction.

LSL actions describe how a Java action is instantiated in the background, and how it behaves based on a set of configuration parameters given in its configuration block (Section 13). While Listing 18 illustrates the general structure of a Java Action with no behaviour, Listing 19 demonstrates how the Java class is instantiated by a corresponding LSL action defined in an LSL study script.

Java actions typically extend an abstract action class from the Actions API and use a set of Java annotations (i.e., markers) to guide their evaluation. For instance, annotation `@LassoAction` marks the `NoOp` class as a LASSO action. The platform automatically identifies all existing actions based on this marker annotation. Optional information can be provided in terms of strings in order to automatically

```java
@LassoAction(desc = "An action with no behaviour")
public class NoOp extends DefaultAction {

    @LassoInput(desc = "a configuration parameter", optional = true)
    public String paramExample;

    @Override
    public void execute(LSLExecutionContext ctx, ActionConfiguration conf) throws
    ↪ IOException {
        // abstraction container (SM)
        Abstraction abstraction = conf.getAbstraction();
    }
}
```

List. 18: General Structure of a Java Action Class (Actions API)

```
action(name:'noOp',type:'NoOp') {
    paramExample = 'hello world'

    dependsOn '...'
    includeAbstractions '...'

    whenAbstractionsReady() { ... }
}
```

List. 19: LSL Action that configures the Java Action in Listing 18

generate a description of available actions for users of the platform (accessible via the web service). Configuration parameters of an action are marked via the `@LassoInput` annotation with a set of optional parameters. The platform scans each class for its public configuration parameters and automatically "injects" the configuration parameters provided in the configuration block of the corresponding LSL action upon initialisation of the Java action class (i.e., here the parameter value of `paramExample` in the LSL action is injected as a field value in the action class `NoOp`). Method `execute` denotes one of the life cycle methods of a Java action class that can be overridden by implementers. By default, all life cycle methods exhibit no particular behaviour. In this example, we have overridden this method in order to demonstrate the access to the "same" abstraction container data structure that is exposed for LSL actions. Based on this container, implementers can manipulate SM/SRMs (i.e., systems and sequences in LSL actions). Furthermore, the execution context of the LSL script is provided in order to obtain supporting services from the platform such as access to the executable corpus and cluster middleware etc. Note that the platform invokes the execute method for each functional abstraction defined.

Apart from the simple features shown, the Actions API is much more feature rich and allows implementers to define their own custom partitioning scheme to signal

Tab. 12.3.: Available Actions in the Research Prototype

Name	Compound	Description
Select		Text-based selection (including IDCS and LQL)
Arena	yes	The arena test driver for executing stimulus matrices. It also defines a set of measurements including code coverage measurements (JaCoCo), mutation score measurements (PIT) as well as dynamic call graph construction (tracer)
Nicad		Code clone detection based on NICAD [54]
EvoSuite		Automated unit test generation using the heuristics-based test generator EvoSuite [84]
Randoop		Automated unit test generation using the random test generator RANDOOP [184]
JaCoCo		Code coverage measurement using JaCoCo
Pitest		Mutation testing using PIT
Rank		Integrates SOCORA's non-dominated sorting for multi-criteria [143]
Test		Runs JUnit test classes directly via Maven's surefire plug-in
Copy		Copy projects of systems
Unpack		Unpack artefacts
Debug		Prints interesting information to support the debugging of actions

the scalability of the action in hand. Java actions can also implement their own strategies to create new abstraction structures.

12.5.2 Predefined Actions

Table 12.3 gives an overview of the predefined actions offered by the research prototype LASSO through the integration of non-trivial tools. These support reoccurring analysis steps, some of which have already been covered in the overview of LSL in Chapter 10.

12.6 Sequence Execution Engine

In order to execute SMs, the arena test driver integrates a sequence execution engine (SEE) that implements and evaluates SSN sheets described in Section 4.2. SEE is designed to *extract* sequences into a well-defined sequence data model that can then be *instantiated* and *executed* on certain (adapted) Java classes (i.e., systems) that are referred to in the SM. It is able to extract sequences from two basic representations:

(1) spreadsheets in SSN (i.e., spreadsheet documents or CSV sheets), and (2) JUnit test classes. The instantiated sequence model and a concrete SRM implementation are used to *store* any observed records including actuations (i.e., exhibited output values) as well as optional measurements.

12.6.1 Extracting Sequences

While a custom spreadsheet parser is used to parse sequence sheets represented in tabular form (either Excel file or CSV file), the JavaParser framework [124] is used to parse and derive sequences from JUnit test classes and their declared JUnit test methods. Each spreadsheet defines exactly one sequence (cf. Section 4.2), while each test method identified in the set of test methods of a JUnit class (i.e., annotated with `@Test`) depicts a single sequence.

Based on SSN, each cell in a sheet contains an expression which is evaluated by the sequence sheet parser. It detects whether the value of a cell contains a cell reference (e.g., *A1*), a constant value (e.g., 5), a (Java) class reference or a method name.

Both extraction methods share a common process model. After parsing, each method invocation of the extracted sequence is passed to a resolution step in which all ingredients of a method invocation are extracted and represented by the abstraction of a sequence in the execution engine. The resolution step involves the identification of the "owner" of a method invocation (i.e., the callee) as well as the resolution of all input parameters passed to the invocation. Internally, the execution engine uses the "symbol resolver" component of the JavaParser framework [124] to resolve any identified method references.

12.6.2 Execution and Observation

Having extracted sequences, the next step is to instantiate each sequence on each available Java class as defined in the SM. This step usually involves the adaptation of Java classes, since we may need to compute matching bindings between the expected interface of the functional abstraction (or class) in hand and the current class (Section 9.3.2). Depending on the configuration settings, the adaptation engine that runs as part of the arena test driver returns one or more adapters which are then instantiated on all the extracted sequences.

Instantiated sequences are invoked using a meta-programming approach. Since we use Java for the implementation of the arena test driver, Java Reflection is used to actually invoke the (underlying) methods of the class under test ("behind" the adapter class).

The resolution and execution process of each class under test is controlled. It is isolated from other class executions in order to avoid inconsistent states. As in classic unit testing frameworks, after each execution of a sequence on a class instance, the state is reset. Several isolated sequence execution processes can run in parallel, thereby taking advantage of the multithreading capabilities of modern machines (i.e., horizontal scaling), in addition to the vertical scaling provided by LASSO (Section 12.2).

As part of their execution, additional measurements can be attached to sequence executions by adding custom "classloaders". Using custom classloaders, the arena test driver supports the generation and loading of mutants, the measurement of code coverage in order to measure the boundary of classes and their software components (based on scope definitions) as well as the construction of dynamic call graphs based on Java agents that manipulate the classes loaded.

12.6.3 Storing Observational Records

Once a sequence model instantiated on a Java class is executed, all observations are stored as records in the model which is linked to the SRM model. Observations include the behaviour exhibited by Java classes for each method invocation as well as optional measurements as described above. After successful execution, the records are inserted into the transactional database of the platform according to the serialisation format in Section 6.6.1 and the database schema described in Section 6.6. At this point, all observations obtained in the arena are ready for script-driven as well as data-driven SRM analysis.

12.7 Extensibility - Integrating Tools and Techniques

The prototype's architecture has been designed with extensibility in mind in order to offer a platform for dynamic (as well as static) analysis services. So in addition to the built-in, predefined LSL actions and their analysis capabilities, LASSO is able to integrate external tools and techniques in various ways. As mentioned in Section 10.3, since LSL actions are designed as reusable, composable units of work, they serve as natural extension points in LASSO. To this end, the platform offers a well-defined *Actions API* as described before. New actions can be derived from existing ones like actions which depend on build automation. For example, the existing actions that use Maven can be extended in order to integrate tools via Maven's plug-in mechanism. Alternatively, any externally available tools may be integrated and invoked (cf. "native test drivers"). Preferably, tools can be provided to LASSO actions in terms of downloadable container "images" through LASSO's

DOCKER repository, or alternatively, created through custom container configurations (i.e., "Dockerfile"). As a consequence, LSL actions can import any valid DOCKER images and run commands on the tool and applications it contains.

13 Search and Curation

In the following two chapters we demonstrate how four specialised analysis services can be built on top of the LASSO platform to provide completely novel and improved solutions to practical engineering problems (thereby demonstrating the validity of Hypothesis 3). In this chapter, we present two applications to (1) support software reuse activities, and to (2) support the automatic curation of live data sets of executable software systems for mining activities and experimentation.

First, we present LASSO SEARCH, an improved code search service that aims to facilitate software reuse. Second, based on the rich, multi-criteria selection capabilities of the observatorium, we present LASSO CURATE, a service that can be used by engineers and researchers to automatically curate a corpus of executable software systems.

13.1 LASSO Search - Behaviour-Aware Reuse Recommendations

As numerous studies have shown in the context of software reuse [149], software developers actually spend a significant amount of time querying software repositories to try to find existing software that could help them build their software applications more quickly, whether it be code snippets, individual classes/methods or fully-blown libraries [212, 204]. However, the processes and search technologies used to do this today tend to be rather ad hoc and unreliable [30, 31]. This is particularly so for code search engines that aim to help developers find existing systems that deliver a certain functional abstraction they have identified a need for. Since they usually rely on text-based selection technologies, such search engines are unable to actually check whether the returned systems are functionally equivalent to (i.e., deliver) the desired behaviour of the sought-after functional abstraction.

Since LASSO offers a scalable approach to the dynamic selection and comparison of software based on systematic behaviour sampling (Chapter 9), the platform naturally lends itself for use as a dedicated code search engine to provide reuse-oriented recommendations like classic test-driven code search engines. LASSO SEARCH, therefore, exploits LASSO's selection capabilities and the arena to provide a behaviour-aware code search engine which is accessible in three ways —

- *LASSO Web Service/Frontend*: a web-based API and frontend using reusable LSL script code templates (i.e., analysis pipelines) that encode existing or custom search strategies,

- *Code Search Frontend*: a Google-like search interface (i.e., web-based GUI) that is specifically tailored to the needs of code searches,

- *IDE Recommender Plug-In*: a plug-in to a popular (Java) IDE (IntelliJ IDEA [127]) to provide proactive (background agent-based) or reactive (user triggered) reuse recommendations (similar to CODEGENIE [154] and CODECONJURER [120]) tightly integrated into the developer's workflow.

Unlike existing code search engines, LASSO SEARCH offers two basic modes for querying software systems of interest: (1) users can use LASSO's LSL as a novel *dynamic query language* (i.e., programmable model) to gain full, fine-grained control over the search process by leveraging its full capabilities, and (2) users can choose to use "classic" frontends that abstract from the underlying LSL representation by providing a simple, traditional query language to formulate software search criteria. While humans can directly write LSL scripts to formulate queries in the former mode, automatic code generation based on LSL script templates is used in the latter mode to search for software systems of interest.

Like any other analysis pipeline in LASSO, LASSO SEARCH is driven by existing and adjustable pipeline scripts in LSL that can be (re)used to find reusable systems composed of Java classes and methods. Whereas the default LASSO interface (i.e., web-based API and GUI) is agnostic to the intention (i.e., usage scenario) behind the submitted LSL scripts, the search frontend as well as the recommender plug-in were specifically developed to "hide" the richness of LSL scripts. They simply use LSL scripts as "search templates" to encode user-supplied reuse criteria in an alternative way. Optionally, of course, users can also directly supply their own modified versions of LSL scripts instead and use them to query for systems of interest.

The service provides several enhancements and contributions that go beyond existing state-of-the-art code search engines. These increase the chance of finding suitable software systems, locating their artefacts, assessing their relevance and adapting them to be fit-for-purpose [176]. Moreover, being able to discriminate systems based on their exhibited behaviour in the arena, and to measure custom engineering goals (Chapter 5) as part of the search process makes it possible to take the quality of systems into account when conducting code searches. LASSO, therefore, provides a rich infrastructure of "tools" to address some of the core obstacles to software reuse in practice [157, 31].

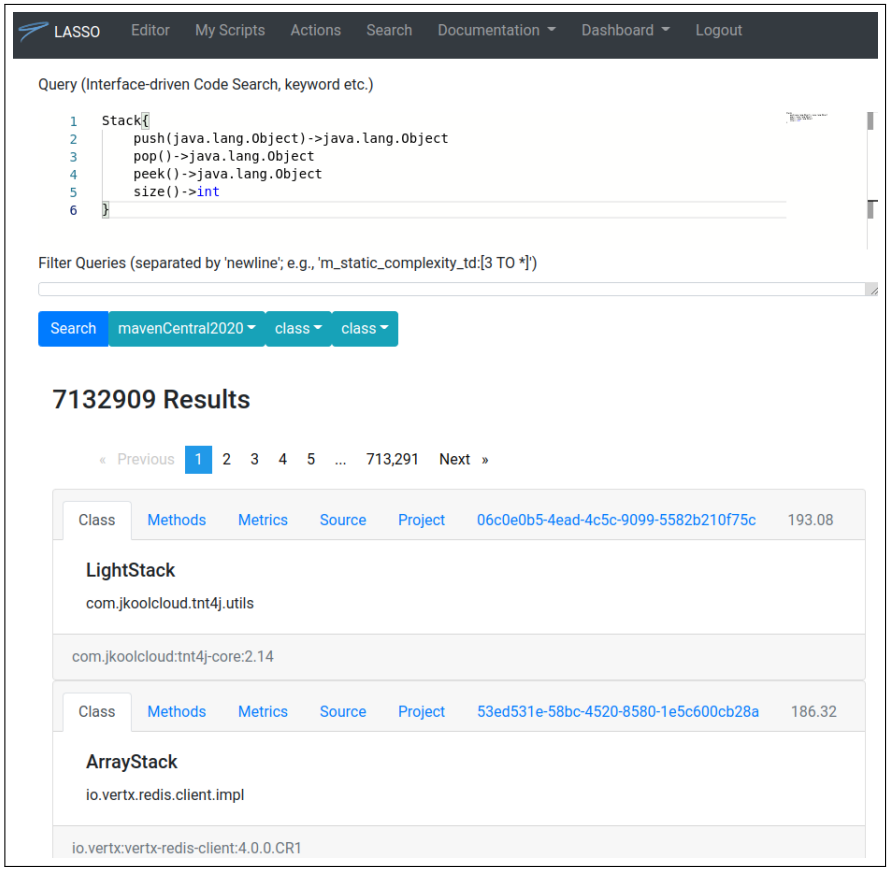


Fig. 13.1.: LASSO SEARCH - Web Frontend (IDCS Query in LQL Shown)

13.1.1 Search Frontend

In order to provide a "classic" search frontend for LASSO SEARCH, we extended the default web-based GUI presented in Section 12.1.6 as shown in Figure 13.1. The layout is motivated by state-of-the-art general purpose search engines and is basically split into two parts, a query form and a pageable results view.

Since the selection criteria offered by the observatorium directly translates into individual reuse criteria, the query form of the frontend is specially customised. It supports text-based queries based on LQL including IDCS, keyword-driven selection, but more importantly it supports LSL as a dynamic query language. In other words, the search frontend basically supports all selection strategies of the observatorium. These include all the selection criteria directly supported by LASSO as well as custom selection criteria (i.e., custom scope-aware measurements) formulated by users via individual LSL scripts.

In the case of text-based queries, the results are displayed immediately, since returned Java classes/methods are retrieved directly from the index of the underlying

corpus. LSL queries, on the contrary, may not immediately return results if they encode search strategies like test-driven selection that involve some non-trivial analysis. Instead, the search frontend returns a unique identifier (i.e., URL) from which the search result can be obtained at a later point in time in a similar way to MEROBASE. In the background, an LSL query is handled like any other LSL script and put into the user's workspace from which all (intermediate) files etc. can be obtained in addition to the information presented in the results view of the search frontend. Users can pick from existing selection strategies that are encoded in LSL or customise them to their needs. The query editor inherits all the properties of the LSL editor (including code highlighting etc.). Details about these LSL templates are discussed in the subsequent subsections.

As shown in Figure 13.1, returned Java classes or methods appear as lists of cards, each of which displays numerous properties about the class/method. Properties include the basic details about the code unit in terms of its name, Maven URI, source code, list of methods declared as well as external links to its artefacts.

13.1.2 Code Recommendation Plug-In

When using the web-based search frontend, users have to integrate desired reusable software systems and their software components manually in their local development project. LASSO's code recommendation plug-in, which offers tighter integration into the developer's workflow, is discussed in this section.

Code recommendation systems essentially build on code search engines by performing more sophisticated searches in a less obtrusive way. LASSO provides the basis for more sophisticated code recommendation techniques since it provides access to many more properties about the reuse candidates, thereby improving transparency.

Figure 13.2 demonstrates LASSO's search plug-in for the popular IDE INTELLIJ. It interfaces with LASSO's code search engine to provide reuse-oriented recommendations (cf. [123]) as part of the developer's development workflow. The plug-in offers both proactive and reactive reuse recommendations that are specific to the current development context (e.g., the files currently viewed in the editor). In case of proactive recommendations, a background service (i.e., agent) running as part of the developer's IDE, continuously runs test- and/or code driven searches when modifications are made to tests or functional code units. This approach provides an unobtrusive way to recommend reuse alternatives, even when the developer is not actively searching for reusable candidates.

The rich selection criteria supported by LASSO essentially facilitate quality-aware recommendations inside the IDE of the developer, and remove the need for devel-

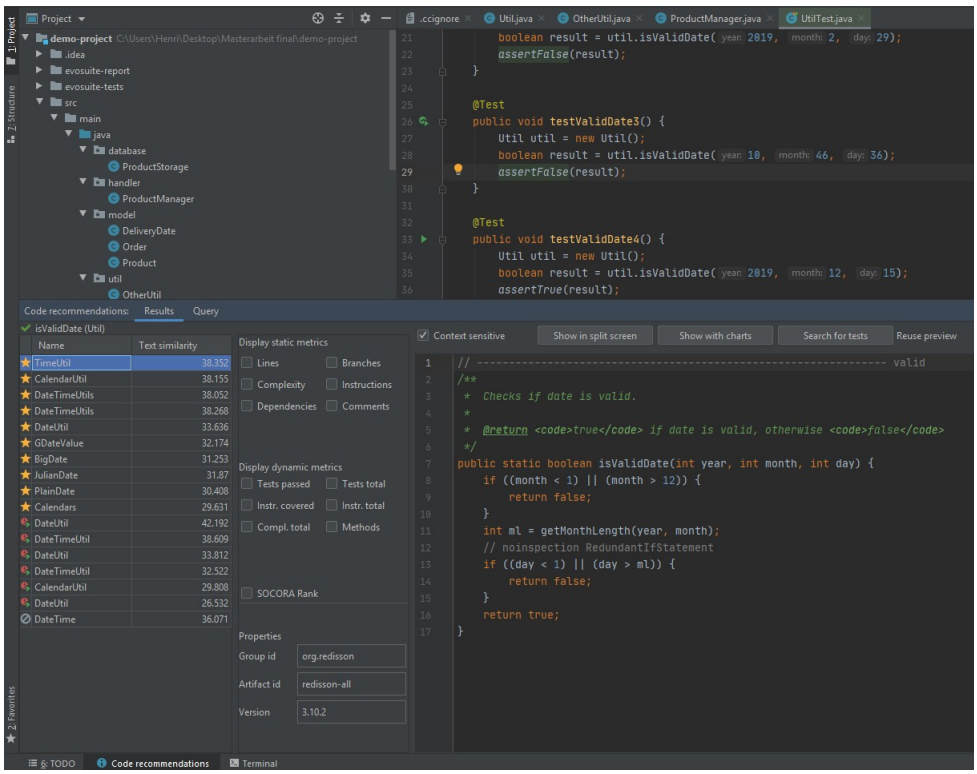


Fig. 13.2.: LASSO SEARCH - Recommender Plug-In for INTELLIJ IDE [156]

opers to spend time on writing queries or sample implementations for test- and code-driven searches. Alternatively, users can opt to actively trigger code searches within the IDE, in case the developer does not want to receive recommendations automatically. An extended overview of the features that the plug-in offers is provided in [156].

Internally, the plug-in interfaces with LASSO's platform via its web-based interface. To submit queries, the plug-in uses LSL as a dynamic query language based on predefined LSL script templates that are filled out with the reuse criteria automatically inferred from the opened files in the editor (i.e., test class or system class code). The plug-in, therefore, also covers the wide range of selection criteria supported by the platform.

13.1.3 LSL as a Dynamic Querying Language

The main novelty of LASSO SEARCH is that its design principles are inspired by established principles of big data platforms. LASSO's basic infrastructure is built to support a high degree of selection flexibility and automation, and allows users to (re)define (search) workflows using a dedicated DSL that combines static and dynamic semantic code search capabilities with analysis services. Selection of

systems is a first class citizen in the design of LSL. An important and social aspect of LSL is that scripts written in the language encapsulate entire selection strategies, and can thus be reused, shared and modified by third-party users. Using a small number of LSL actions, sophisticated selection strategies can be implemented in a relatively straightforward way. Instead of the rigid, static nature of existing code search engines, LASSO SEARCH shifts the focus to the search flow and allows it to be adapted to include advanced, individual reuse criteria important for the (reuse) task at hand.

LSL facilitates the design of new or enhanced retrieval strategies since existing actions in the pipeline can be "reassembled" or removed and new ones can be added. This not only continually improves the precision/recall of current selection strategies, but also increases the range of information that can be mined from them. As mentioned above, users have the option of gaining full control of the selection process or to use simpler, default selection strategies that further simplify the query formulation. LASSO SEARCH basically supports developers at any stage of their development progress. In the recommendation plug-in, for instance, the domain-specific knowledge wrapped up in existing code (i.e., code skeleton or partial implementation), and the actuations that specify the behaviour of the sought-after functional abstraction, can be used by LASSO SEARCH to offer various selection options, thereby covering the full range of search styles, from speculative searches to definitive searches (cf. [117]).

Test-Driven Search

This section shows how selection strategies can be encoded in LSL pipelines and how straightforwardly they can be extended to encode user-specific reuse criteria. The LSL script in Listing 22 (Appendix) shows how the test-driven selection process (Section 9.1) can be encoded as an analysis pipeline in LASSO based on the stack abstraction. Apart from the two typical steps of an IDCS (i.e., a select action to obtain a preliminary collection of systems and a filter action to remove unsuitable systems), it also contains two additional actions to filter candidates based on the presence of code duplicates and to finally rank candidates according to their relevance.

Whereas the select action allows text-based queries and filters based on the information available in the underlying corpus of LASSO, the arena test filter allows the specification of stimulus sheets to further improve the precision (i.e., relevance) of the candidates returned based on their exhibited behaviours. In this case, any candidates that do not match the expected output values in the stimulus sheet are rejected. Users can provide their own rejection criteria by simply modifying the `whenAbstractionsReady` block. Whereas existing test-driven code search engines

like MEROBASE apply strict selection criteria (strict behavioural matching based on a set of tests), LASSO SEARCH allows "best-effort matching" where users can specify their individual criteria (e.g., allow the passing of a subset of the tests to measure functional similarity).

The declaration of additional actions allows advanced filtering options to be specified like the removal of code duplicates (here type-2 code clones). In other words, users have the ability to use LSL as a dynamic (programming) language to flexibly express and enforce their individual reuse criteria, while improving the relevance of the search results.

Since code searches typically involve some sort of "ranking" based on relevance, the last action enables users to change the default sorting of candidates. In this case, we demonstrate the integration of the SOCORA ranking method as a LASSO action that provides a non-dominated sorting of multiple measures based on user preferences [143]. The action formulates ranking criteria based on the analysis attributes (i.e., observational records) that are stored in SRMs. Alternatively, users can formulate their own ranking strategy in terms of a plain LASSO action, for instance. Note that text-based selection also offers the capability to sort the candidates that are returned from the index of the executable corpus based on SOLR's ranking capabilities.

Code-Driven Search

LASSO is able to boost the discriminating power of test-driven searches by supporting the automated creation of additional test sequences and using the created data to compare the behaviour of the candidate systems. The automated test generation capability of LASSO can support code-driven selection (CDS) on top of test-driven selection (Section 9.2). To the best of our knowledge, no similar technique has been proposed or implemented to date. Using CDS, the test-driven selection process can be almost fully automated, freeing users from manually writing tests. Variants of CDS can be used —

- *One-To-Many*: to find alternative implementations based on a given (reference) implementation of a given functional abstraction,
- *Many-to-Many*: to improve precision by performing cross-running test sequences in the arena on alternative implementations and identifying discrepancies (using appropriate SRM configurations).

Listing 23 (Appendix) shows a comprehensive LSL script that reflects a (one-to-many) CDS for a certain (pre-known) stack implementation (identified via its unique identifier in the underlying corpus). The pipeline for one-to-many CDS consists of

five actions in this particular case. However, it can be reduced to four actions by rejecting the optional code duplication filter (i.e., `clonesAlt`).

The first action selects a known stack class from the executable corpus for which, in the second action, test sequences are generated using the EvoSuite action. We refer to the single class implementation in a one-to-many CDS as the reference implementation. Thereafter, the third action, `selectAlt`, searches for alternative class candidates "by example" based on an IDCS query formulated from the reference class implementation. Then, action `clonesAlt` attempts to reject any class duplicates (including the detection of clones for the reference implementation). Finally, action `arena` runs the generated tests for the reference implementation on the alternative classes in order to establish whether they exhibit the same behaviour as the reference implementation class (i.e., to check whether their behaviour is functionally equivalent). Note that the custom selection criteria that are applied to the resulting SRM have to be provided by the user as for test-driven selection.

13.1.4 Use Cases

As mentioned above, the search pipelines for TDS and CDS are provided as script templates that can either be modified by users directly or filled in automatically by search frontends (i.e., web-based search GUI or recommendation plug-in).

The presented search scripts demonstrate that only a minimal set of simple, recurring actions is required in order to formulate rich, custom, behaviour-aware search strategies. We believe that even new users can quickly learn and use the provided search scripts and modify them to their individual needs. Even if they do not want to use LSL as a querying language, they can use the simpler, traditional variants of LASSO Search in the classic, form-driven way (search frontend), or the plug-in integration to leverage the code search capabilities provided.

Even when formatted in a "pretty" (i.e., human-readable) way, the code size of LSL scripts for TDS and CDS is rather small, showing the expressiveness and efficacy of the LSL language as a dynamic querying language. At the same time, LSL provides users with significant flexibility for developing new search strategies by either modifying existing actions or by adding additional ones in order to encode custom reuse criteria or custom search strategies. For example, the number of adapters generated in TDS for a certain class candidate can be set by a single parameter in the configuration block of the arena action.

To the best of our knowledge, none of the state-of-the-art code search engines offer a dynamic query language that offers observation-based services and integrates state-of-the-art tooling. They are mostly limited to rather "static" query languages

that only allow for a limited set of predefined selection criteria. In other words, the user has no, or limited, control over the search process.

Since search strategies are encoded as LSL scripts, users can exploit the full potential of LSL and the analyses services provided by the observatorium. Users, for example, have the opportunity to access the results of past LSL script executions in other scripts by simply referring to them using special URIs (see Section 10.2.3). This enables the structuring of search strategies into smaller subscripts (e.g., divide-and-conquer) and enables the reuse of past results.

Measurement and Comparison

SRMs can store attributes of any kind, so analysis criteria based on virtually any (measurable) engineering goals can be formulated. For dynamic measurements, the arena is able to obtain any measurable properties and can conduct dynamic measurements of each system's execution, even based on behaviour-aware system boundaries (cf. Chapter 5). This includes, among other things, functional and non-functional properties such as performance measurements and direct/indirect metrics based on (dynamic) call graph analyses (e.g., code coverage measurement).

The collection of measurements in SRMs is useful for two reasons – (a) they can be used as selection criteria to enable quality-based reuse [157, 31, 27], and (b) they reveal differences and interesting comparisons between the systems involved to facilitate decision-making in the reuse process (i.e., is a certain candidate fit-for-purpose, hence is the potential reuse cost-effective ? [23] Are there any side effects? [163]). Augmenting the available knowledge about tested implementations not only increases their transparency, it also allows users to reject undesired systems based on properties such as code clones or undesired structural design. Moreover, by gathering more information about tested systems, interesting insights, relationships and discrepancies about the tested systems and their functional abstractions may be revealed.

Dynamic metrics not only provide increased transparency for selecting the "best" implementation from a large list of true positives, but also contribute to cases in which no true positives are found. In this case, the re-user may use the gathered properties about the implementations to re-design the test sequences in TDS.

Filtering Options

As we have demonstrated in Listing 22 and 23 for TDS and CDS, various filters can be specified in a variety of ways to increase the relevance of the returned candidates.

There are three basic ways to specify filters in LASSO SEARCH, including —

- "static" filter queries based on properties present in the underlying corpus using the select action (Section 8.2),
- advanced (dedicated) filter actions like code clone actions (Section 16),
- custom plain filtering behaviour specified in code blocks of existing or plain LSL actions (cf. Section 10.3).

These filtering options can either be used independently or they can be combined into a hybrid filtering strategy. Such filtering capabilities can be used to improve the efficiency of the search process (e.g., rejecting undesired candidates such as code duplicates early in the process), or to further increase the diversity of the candidates (e.g., rejecting candidates based on additional measures etc.). Moreover, users may opt to integrate their own (advanced) filter action by integrating an existing tool or approach and exposing it as a dedicated filter action.

Note that filter queries also facilitate type-driven queries that allow users to retrieve classes that are of a certain type (i.e., classes that extend a certain super class or a certain interface in their type hierarchy). This is especially useful in scenarios in which users are interested in certain application domains. If an Android developer, for instance, is interested in getting types of an Android `Activity`, a corresponding type-hierarchy aware filter can be specified to only return matches that are explicitly of type `android.app.Activity` [9].

Virtually any information such as measures obtained in the execution of LSL actions (e.g., arena measurements such as software metrics) can be used to filter candidates in the result set, thereby allowing their relevance to the specific reuse objective to be reassessed. In other words, LASSO SEARCH provides a flexible means to formulate and express filtering-based reuse criteria, and to optionally encode custom relevance-based scoring methods to identify "best matches" and to "rank" matches according to their relevance.

Executability, Testability and Environments

The unified repository model realised by the executable corpus of the observatorium improves the success rate in obtaining executable systems, and hence improves the selection, adaptation and integration of reuse candidates in the developers' application. The automated build script synthesis based on the popular Maven build tool enables users to easily integrate reusable candidates in their own Java projects (Section 7.4).

Since behavioural observations about systems are often sensitive to the surrounding environment, flexibility also applies to the definition of (controllable) execution environments in which systems are exercised. Instead of defining each execution

environment manually, LASSO takes advantage of containerisation technology to allow environments to be automatically selected and created "on-the-fly". To the best of our knowledge, none of the proposed test-driven search engines have investigated the target run-times of reused implementations, even though this allows users to receive important feedback about the desired version compatibility (e.g., Java Version).

As we have demonstrated in Listing 22 and 23 for TDS and CDS, users are free to specify their target execution environments in order to assess whether potential reuse candidates meet their requirements. Moreover, TDS and CDS pipelines can be simply modified or extended to check whether systems do execute in multiple environments and whether they exhibit the same behaviour (e.g., comparing SRMs).

13.1.5 Potential

We believe that LASSO SEARCH has huge potential to further enhance software reuse. As well as the "recommendation tool" model for reusing LASSO SEARCH in which it is tightly integrated into the IDE of the developer, it is also possible to use LASSO SEARCH to support "continuous reuse" [139]. Since LASSO is compatible with modern continuous integration platforms, its dynamic analysis capabilities can be exploited at opportune moments for continuous code recommendation. For this, we envision integrating LASSO SEARCH into continuous integration (CI) pipelines [78] to allow developers to receive recommendations in an unobtrusive manner based on text-based, TDS and CDS queries that are generated and submitted automatically. This service can be used to identify reusable candidates when the CI system is idling, and to provide fast feedback to developers if existing systems already exist. This opens up the possibility for companies to discover existing functionality they were unaware of in early phases of their projects, thus making it possible to reduce overall development costs.

LASSO's flexible pipeline design enables the benchmarking of new and existing search strategies. Since search strategies are encoded in LSL scripts, LASSO SEARCH enables search strategies to be assessed with respect to important evaluation criteria (i.e., precision and recall). Here users can compare the SRMs produced by each search script using the observatorium's data analytics layer. Benchmarking can be used to improve existing search strategies and to assess new ones. For example, the precision of IDCS can be assessed by looking at the number of false positives when evaluated in the arena based on a set of test sequences.

Finally, since systems and test sequences are closely related in practice, there is great potential to further leverage the domain knowledge prevailing in existing tests (i.e., test classes). Facilitating the explicit discovery of tests enables further search

strategies such as backward searches for the class(es) under test. A major challenge here is to identify the actual class(es) under test. Finding test classes opens up a new dimension of use cases. For example, creators of systems may be interested in how their systems are "reused" by others. Since re-users may specify tests for their (adapted) reused systems as well, the original maintainer of a system (e.g., in the context of Open Source) can find tests that were explicitly written for his/her system in order to make the system more robust or to identify additional potential use cases for the functional abstraction.

13.2 LASSO Curate - Automatically Curated Data Sets

Up to this point, we have discussed how the LASSO platform can be used to enhance several aspects of code search engines in order to facilitate software reuse tasks. However, the technology, particularly its selection capabilities, are also useful in several other engineering tasks and activities that rely on collections of software systems that possess certain properties (i.e., meet certain engineering goals).

In the following we present LASSO CURATE, an analysis service built on top of LASSO that automatically curates data sets of executable software systems. Here, selection criteria encoded in LSL scripts translate directly into curation criteria. Even though LASSO CURATE basically addresses the same initial challenges as the basic creation of an executable corpus (Section 7.2), its selection requirements go beyond them.

A major step towards curating data sets automatically is to construct software corpora using a generic, script-driven environment which supports the integration and realisation of custom analyses for the task at hand. Palsberg and Lopes [185], for instance, propose a corpus of executable Java software systems that is integrated with a catalogue of established tools (e.g., for code analyses and measurements).

Similarly, but going one step further, the creators of XCORPUS [68] introduce their vision of a "*live data set*". A live data set is an executable corpus which is (dynamically) "distilled" from a single, underlying corpus of executable software systems whose analysability can be automatically extended in a flexible manner to support different mining and analysis tasks. The realisation of live data sets has two core requirements —

1. The creation and evolution of a *single, underlying corpus of executable software systems* from data sources of interest (i.e., software repositories),

2. The provision of an *automatic curation capability* to select, analyse and compare executable software systems of interest.

LASSO already meets the former requirement by the way its executable corpora are created. The latter requirement, on the other hand, is met by LASSO's arena that facilitates the definition of custom selection strategies and rules to generate custom live data sets on-the-fly. Moreover, LSL offers a unified and extensible script-driven environment in order to (1) provide a dynamic query language for curators, and (2) to "plug-in" custom tooling and to store live data sets.

Interestingly, the arena plays a special role in the context of live data sets. It is both a "consumer" of live data sets (i.e., it is populated with a collection of systems), but at the same time it can also be used as a "producer" of live data sets (i.e., a set of systems collected as a result of analysing the SRMs produced by the arena).

13.2.1 Behaviour-Aware vs Behaviour-Agnostic Curation

In general, there are two basic types of curation criteria available that depend on behavioural properties of software systems. They basically differ in terms of whether one or more implementations of the functional abstraction of interest must be present —

- *Behaviour-aware*: Test-driven selection is conducted for the functional abstraction of interest (to obtain many functionally equivalent implementations),
- *Behaviour-agnostic*: The actual behaviour of software systems exhibited in the arena is unimportant as long as they satisfy the properties of executability and measurability.

The ability to perform behaviour-aware curation is the foundation for many advanced techniques that rely on a particular functional abstraction such as automated code and test enhancement. Techniques such as automated program repair [135] start by finding a suitable fragment of code that can be woven into an existing implementation and then subsequently evaluate the semantic suitability of the resulting code.

Behaviour-agnostic curation criteria, on the other hand, only requires the presence of "any" (often non-trivial) behaviour, and the actual functional abstractions realised by the systems are not of interest. However, the properties of executability and measurability need to be met by selected systems. Behaviour-agnostic curation criteria, therefore, are often needed to validate hypotheses and tools that depend on the executability of software. An example are benchmarks of software engineering tools that rely on a set of study subjects (see Chapter 15).

Approach

In order to realise LASSO CURATE, LSL is used as a dynamic query language, but for the purpose of formulating individual curation criteria, and to automate data set curation tasks (similar to the formulation of reuse criteria in Section 13.1.3). Behaviour-aware selection criteria can be encoded by writing LSL pipelines based on "classic", test-driven selection using manually-written test sequences (e.g., sequence sheets) as illustrated in Listing 22. Classic, test-driven selection in the spirit of test-driven code search engines, however, is limited and often impractical if larger data sets are needed. Another way to obtain behaviour-aware collections of systems is to use CDS for selection as illustrated in Listing 24 (Appendix). In this case, a set of alternative implementations is harvested for a given reference implementation that represents the functional abstraction of interest for which test sequences are automatically generated. This is a promising approach, because in many usage scenarios, (human) curators do not need to know (cognitively) the actual functional abstraction that maps to the exhibited behaviour of the reference implementation. All that is required in these circumstances is that the alternative implementations are functionally equivalent to the behaviour of the reference implementation.

Behaviour-agnostic selection criteria can be applied by designing LSL pipelines that simply focus on assessing whether the properties of executability and (optionally) measurability are met by a collection of (pre)selected software systems. In this case, we need a way to stimulate the system of interest with some execution scenario that (hopefully) confirms the property of executability. A straightforward way to achieve this capability is, again, to obtain automatically generated test sequences. Any software system that cannot be stimulated in the arena is removed in the resulting SRM. As an example, the LSL pipeline in Listing 24 (Appendix) can be cut down to only contain two actions, one for selection from the index of the executable corpus, and one for automated test generation. The resulting LSL is illustrated in Listing 25. Note that this pipeline also includes one additional filtering step which is the dropping of code duplicates.

As with any other LSL pipelines, curators can exploit all (selection) capabilities of the observatorium in order to obtain live data sets with high precision. The data sets selected through the observatorium can be exported in several ways to either post-process them (e.g., in the data analytics layer) or "use" the systems and their artefacts for the intended purpose (e.g., experimentation). Advanced analysis of software collections in the data analytics layer may involve answering (research) questions in a data-driven way (like BOA), or feeding machine learning pipelines to derive new knowledge (cf. mining software repositories).

Executable software systems can be "exported" into a collection on a local system by retrieving their (Maven) artefacts. This is facilitated by the fact that curators can access the artefact repository provided by the executable corpus systematically. Maven artefact coordinates are stored as part of SRMs and can be used to "download" the artefacts for use in external tools.

13.2.2 Data Set Properties

In addition to the properties that single systems in a live data set have to possess, there are often second-order properties of live data sets that must be met as well.

Subjects - Level of Granularity

A basic question in a curation goal are the subjects of the curation process – what is actually being curated (i.e., which level of granularity is required)? By default, for software systems we assume the class and method level of granularity. In manually curated data sets in the real-world, however, curators often collect and prepare software projects that contain one or more software systems (e.g., SF110 [83]). The curation of entire software projects is possible in the observatorium by using a special configuration of the SELECT action that first samples single classes of interest and then obtains all their project-related classes (using a follow-up query based on the Maven coordinates that indicate the project of the selected class).

Sampling Strategies

Curation criteria may implicitly assume or explicitly define a certain sampling strategy for a certain purpose. In software experimentation, for instance, a typical sampling strategy [20] is to sample software artefacts by their popularity (e.g., popularity in a software repository).

Another strategy to cope with generalisability of analysis results is to apply random sampling strategies. Random sampling is explicitly supported in the observatorium and achieved in the preselection phase of retrieving software systems from the index of the executable corpus. By default, software systems (i.e., Java classes or methods) are returned sequentially, in the order of their text-based relevance (determined by text-based queries). The ordering criteria can also be customised by users in LSL.

Diversity

Many (test-driven) curation criteria include second-order properties that a collection (i.e., curated set) of software systems must satisfy. The most prominent example is

when a collection of software systems should ideally exhibit a high level of diversity with respect to (1) implementational distinctness (assuming functionally equivalent systems), or (2) behaviour in terms of functional abstractions. For the former, as explained in Section 14.1, implementational distinctness can be increased in a variety of ways, including code clone detection, the analysis of indicator metrics or metadata. This criterion is often included in behaviour-aware selection criteria that aims to retrieve sets of diverse implementations of a certain functional abstraction. For the latter, the total number of functional abstractions in a live data set can be increased by the analysis of SRMs produced by the arena. Assuming sets of similar interfaces, the analysis of behavioural relationships based on sets of actuations can reveal the existence of duplicates of functional abstractions.

14 Exploiting Diversity

This chapter presents two additional analysis services built on top of LASSO to further confirm Hypothesis 2. The two analysis services provide enhanced and novel approaches for improving state-of-the-art software testing practices, these include —

- LASSO TESTGEN: a service that offers diversity-driven test generation,
- LASSO TESTAMP: a service that offers diversity-driven test amplification.

Both applications are driven by the idea of exploiting the implementational distinctness found in systems harvested from large software repositories. The following sections first introduce the notion of implementational distinctness and clarifies the notion of redundancy in our context, then explain the LASSO TESTGEN and LASSO TESTAMP approaches.

14.1 Diversity

For certain analysis scenarios, a desirable goal is to distinguish between systems that take a truly different approach to implementing a functional abstraction from those that adopt essentially the same approach but might differ in superficial ways, or might not differ at all. Identifying the difference between two implementations is typically achieved using criteria and constraints derived from the system's textual code elements, assuming that software systems are functionally equivalent to the functional abstraction in a pair-wise way.

Implementational distinctness among a pair of systems is defined as follows. Two systems are **implementationally distinct** if they implement the same functional abstraction in an inherently different way. This concept can obviously give rise to a form of similarity measure that could reasonably be referred to as "implementational similarity". However, we use "implementation distinctness" to emphasise that we are interested in low implementation similarity.

Among other things, implementational distinctness may be identified in three main ways, by —

- detecting and rejecting code duplicates using code clone detection techniques [207] (i.e., as an additional analysis step),

- measuring and comparing (weak) indicator metrics such as size-based software metrics (e.g., LOC or CC),
- reasoning over the metadata of systems (e.g., different author).

Based on the functional equivalence relationship (see Section 3.4.1) and the notion of implementation distinctness, two basic forms of redundancy between implementations of a functional abstraction can be defined.

Two or more software systems are said to exhibit **simple redundancy** if they are functionally equivalent. This definition means that a software system is simple redundant with itself. Moreover, any functionally equivalent code duplicate of one software system is redundant with the original software system. Including the notion of implementational distinctness, a collection of two or more systems exhibit **heteromorphic redundancy** if they are functionally equivalent and are implementationally distinct. Simple redundancy is therefore necessary but not sufficient for heteromorphic redundancy.

Once implementational distinctness (or similarity) can be identified for a set of functionally equivalent systems, the property can be applied to obtain second-order properties of collections of systems (including repositories) such as **diversity** and **sparsity** [4]. Diversity obviously increases as the number of distinct systems in a population increases whereas sparsity decreases. The two can therefore be regarded as the inverse of one another.

14.2 LASSO TestGen - Diversity-Driven Test Generation

Tools for AUTG in software testing have matured significantly over recent years, and thanks to their meta-heuristic search algorithms (cf. SBSE/SBST [107, 174]) and static/dynamic program analysis techniques they can now routinely achieve branch coverage levels and mutation scores of well over 70% [83, 187]. However, opportunities for future improvements are limited by the fitness landscape faced by AUTG tools [252, 11], and by their reliance on the availability of one, and only one, implementation of the functional abstraction of interest to apply their analysis algorithms.

A potential way of producing higher quality tests, therefore, is to give them access to more, diverse, implementations of the functional abstraction of interest, and thus more structural and domain information to work with. In [138], we propose a novel, hybrid AUTG approach, DIVGEN (Diversity-driven Test Generation), that combines the mainstream AUTG tool, EVOSUITE, with the search and observational capabilities provided by LASSO in order to generate test sets of higher quality than EVOSUITE

can achieve alone. In the remainder of this thesis, we refer to the LASSO-based realisation of DIVGEN simply as LASSO TESTGEN (or TESTGEN).

Since the approach combines SBST with search-driven software engineering, it can be seen as one incarnation of a search-based software engineering approach [15]. Note that "diversity" in this context is regarded as the availability of a diverse set of implementations as defined above and should not to be confused with "population diversity" as known in SBST. There it refers to a (desirable) property of evolutionary algorithms [3].

Even though AUTG tools like EVOSUITE and the TESTGEN consume a non-trivial amount of computing resources, they can still be useful in practice. Since TESTGEN is fully automated, and can be run in the background as part of continuous integration pipelines (cf. [78, 139]), it has the potential to significantly enhance the return on investment obtained from AUTG approaches in modern software development processes.

In the following, we first provide an overview of our AUTG approach, and then we discuss its implementation as an analysis service offered by LASSO.

14.2.1 Algorithm

To introduce the basic idea behind TESTGEN, we present and discuss the pseudo algorithm of TESTGEN abstractly, followed by an LSL script that demonstrates a practical realisation of LASSO TESTGEN. More details can be found in [138]. Since TESTGEN is realised for SUTs written in Java, we refer to the SUT as the CUT (class under test) in the remainder of this section.

The basic idea we exploit in our approach is to expand the search space of AUTG tools by leveraging the (extra) domain knowledge "encoded" in alternative implementations (i.e., to exploit more code units produced by possibly different developers [16]). Alternative implementations are harvested from software repositories based on the functional abstraction realised by the CUT. By giving EVOSUITE extra domain knowledge, we hypothesise it can deliver higher quality tests compared to AUTG approaches that are only applied to a single CUT.

TESTGEN uses the existing AUTG tool, EVOSUITE, as part of its approach to generate tests for single CUTs. But in contrast to EVOSUITE's default behaviour, it generates tests for harvested, alternative implementations as well. Note that for theoretical reasons, the actual AUTG used by TESTGEN may be viewed as a black box, and thus may use other AUTG tools as well. This is the reason why we abstractly refer to calls to EVOSUITE as $MonoGen(c, b) \rightarrow T$ which highlights the single implementation focus of current AUTG tools.

Function $MonoGen(c, b) \rightarrow T$ accepts a class under test, c, a time budget, b, measured in minutes, and returns a test set T for class c. Note that the time budget parameter is an important parameter for search-based AUTG tools that employ randomised algorithms. It depicts the time budget that is available for the optimisation algorithm to try to obtain an optimal set of tests. The default time budget proposed by the EvoSuite authors is two minutes.

Figure 14.1 depicts TestGen's test generation approach in terms of a pseudo algorithm. Function $TestGen(c, m, b, a) \rightarrow T, n$ has four input parameters. Parameter c depicts the CUT, parameter m the maximum number of alternative implementations to be retrieved by invoking auxiliary function $Harvest$, parameter b the time budget for running $MonoGen$ on each available implementation, and parameter a the maximum number of adapters to be used to adapt generated tests to the interface of CUT c.

Input: Class c, maximum number m of alternative classes to retrieve, b time budget for $MonoGen$, a maximum number of adapters
Output: Generated test set T_{san} for c, and n, total number of implementations used to generate tests

```
1  Function TestGen(c, m, b, a) → T, n:
2      C_alt ← Harvest(c, m)
3      C ← {c} ∪ C_alt
4      T ← ∅
5      for c_i ∈ C do
6          T_ci ← MonoGen(c_i, b)
7          T_adap ← Adapt(c, a, T_ci)
8          T ← T ∪ T_adap
9      end
10     T_san ← Sanitise(c, T_adap)
11     n ← size(C)
12     return T_san, n
```

Fig. 14.1.: Pseudo Algorithm of $TestGen$

The output parameters of TestGen are a merged test set T_{san}, and the number of actual implementations that were used in the generation process (i.e., 1 plus the number of harvested implementations).

Internally, TestGen executes four basic steps. First, the interface signature of the CUT is extracted in order to retrieve at most m alternative implementations using $Harvest$. Theoretically, $Harvest$ can use any search strategy (e.g., IDCS, TDS etc.) in order to retrieve alternative implementations. Once alternative implementations have been obtained, all available implementations are passed to $MonoGen$ in order to generate tests for them. Thereafter, all the test sequences that were generated are "adapted" (i.e., made compatible) to run on the CUT (cf. $Adapt$ function, see

Section 9.3.2). Note that only the stimuli are adapted to the CUT, the responses are ignored. All tests that could be adapted to the CUT are then run on it in order to filter out those that work on it (i.e., $Sanitise$ function), and to obtain its responses (i.e., to enable regression testing at a later point). Finally, a clean set of tests is returned by TESTGEN. In practice, the final, clean set of tests is a set of test classes (i.e., JUNIT tests) that are generated from the cleaned test sequences, including the actual responses (i.e., outputs) obtained from the CUT at run-time.

14.2.2 Realisation

LASSO TESTGEN is accessible as a reusable analysis pipeline written in LSL (i.e., script template). The analysis service script in Listing 26 (Appendix) presents one possible incarnation of TESTGEN as proposed in [138]. Here the auxiliary $Harvest$ function of TESTGEN (cf. Figure 14.1) is based on IDCS.

In this case, the basic pipeline design consists of five LSL actions that apply TESTGEN to a known CUT that realises the stack abstraction. Accordingly, the reference implementation (i.e., known CUT) is retrieved from LASSO's executable corpus using the first `Select` action. The second `Select` action attempts to return alternative implementations of the reference implementation based on IDCS (i.e., the $Harvest$ function in Figure 14.1). Here an "example query" is constructed from the interface signature of the CUT. Additional filters are defined to exclude abstract and inaccessible classes. Moreover, classes that contain test code, or are deemed to provide internal services (i.e., classes that are part of the Java JDK or analysis services) are excluded as well. Thereafter, the third step executes a filtering action that detects any class duplicates in the set of reference and alternative implementations and removes them accordingly.

Having obtained a set of classes that exhibit "implementation diversity" with respect to type-2 clones, the next two actions of type `EvoSuite` apply the AUTG tool to both the reference implementation and the alternative implementations. Finally, the last `Arena` action receives all implementations and their generated test sequences. It attempts to (partially) adapt and execute them on the reference implementation. Note that any failures are automatically caught and handled by the default failure mechanism provided by the prototype platform. Any alternative implementations for which either no tests were returned or EVOSUITE failed are simply ignored in the arena execution step. Test sequences that were not applicable to the reference implementation are dropped as well. Finally, once a preliminary set of non-duplicate test sequences has been identified, the "amplify" capability of the arena exports the test sequences into JUNIT classes and re-runs them on the reference implementation again in order to sort out test cases that are flaky (i.e.,

test sequences that exhibit non-deterministic behaviour in subsequent runs) or that cannot be run on the reference implementation because of technical limitations (e.g., unresolvable classes etc.). In other words, the last step is to return a "sanitised" set of tests.

Note that the actual test sequence mining part of TESTGEN is realised as a "task" that utilises LASSO's arena (see Section 12.6). The basic idea of the mining algorithm is to attempt to identify test (sub)sequences that can be executed on the reference implementation's class (cf. Section 4.6) with respect to the current adapter generated for the class. As part of the export of JUNIT test classes, the exhibited behaviour of the reference implementation can be included as well in order to support regression testing scenarios like current AUTG tools.

14.2.3 Potential

The pipeline script highlights how each aspect of TESTGEN can be mapped to a reusable analysis pipeline and corresponding LASSO actions. Like the abstract functions defined in TESTGEN's pseudo algorithm, the actual actions in the pipeline script may be substituted by other actions that provide the same "service". For example, the EVOSUITE action may be replaced by some other test generation action, or the filtering action to reject code clones may be replaced by some other tool for identifying a set of diverse implementations (e.g., based on software metric measures, for instance).

Note that there are many other possible ways to realise TESTGEN pipelines in LASSO. Apart from substituting actions, one may use alternative approaches for harvesting alternative implementations (e.g., TDS instead of IDCS to harvest functionally equivalent alternative implementations), different parameter settings for actions or alternative orderings of actions. It is even possible to omit test generation for the CUT.

We believe there is significant potential to fine-tuning the parameters of the actions. The time budgets assigned to $MonoGen$, and the number of adapters to generate for the reference implementation could have a significant impact on the quality of the test set returned by TESTGEN. Moreover, the pipeline can easily be extended to introduce additional processing steps like the minimisation of test sets (cf. test suite minimisation [228]) to further increase the efficiency of the approach. Since TESTGEN relies on the domain knowledge embedded within alternative implementations retrieved from LASSO's underlying corpus, the integration of new data sources will probably further improve TESTGEN in practice (i.e., the more code, the higher the probable level of diversity in the corpus).

14.3 LASSO TestAmp - Diversity-Driven Test Amplification

The AUTG approaches such as EvoSuite and TestGen, require access to the code units of a SUT. Since code units represent their primary input, these approaches can be characterised as white box test generation approaches.

A core limitation of most AUTG approaches is that they are "agnostic" to the functional abstraction the SUT realises. Since they do not take into account the functional "semantics", they typically lack important, domain-specific knowledge. A different way of obtaining high quality tests is to take advantage of potentially existing test sequences that are usually written by developers when developing software. These typically encode domain-specific knowledge (i.e., of the functional abstraction and its behavioural specification). Since these approaches attempt to improve the quality of existing test sequences, they fall under the umbrella term of **test amplification** [58, 136].

This term is used by some to characterise approaches which aim to improve the quality of an existing set of tests by changing them, (or their execution behaviour) and/or by extending them with additional test sequences (e.g., [106, 257, 59]). Improvements may be made either to stimuli, responses or actuations. As such, they often exploit techniques from SBST and behavioural sampling research (cf. Chapter 9). As pointed out by Danglot et al. ([58]), "test amplification is defined as taking as primary input test cases written by developers". Four different flavours of test amplification are identified, depending on whether they exploit code evolution or whether they change the testing code itself.

Using the capabilities of the LASSO platform, we created LASSO TestAmp, a test amplification service that takes in a set of tests and attempts to "amplify" them based on test sequences that are mined from a set of diverse, alternative implementations of the functional abstraction of interest. As a counterpart to TestGen, we refer to our novel, diversity-driven test amplification approach as TestAmp. The approach falls under the AMP_{add} category of test amplification approaches, since it amplifies tests in a "black box" way by "Adding New Tests as Variants of Existing Ones" [58, 136]. To the best of our knowledge, TestAmp is the first implemented amplification approach in this category.

Note that the relationship between TestGen and TestAmp is similar to the relationship between CDS and TDS. While TestGen and CDS require the code units of a system as their input (i.e., white box approaches), TestAmp and TDS operate exclusively on test sequences. Similarly, whereas CDS simply "calls" TDS to return alternative classes after it has generated tests, (pseudo) function $TestGen$ simply

calls $TestAmp$ to amplify a test set after it has generated it. In other words, the basic difference between TESTGEN and TESTAMP is that TESTGEN carries out an additional step in which it uses a (white box) AUTG tool to generate a set of tests that are then "amplified" using TESTAMP.

14.3.1 Algorithm

$TestAmp$ is a pure amplification function which only requires a test set for the functional abstraction of interest in order to create new tests. Note that $TestAmp$ does not require any access to an implementation of the functional abstraction of interest, unlike $TestGen$ for AUTG, so it is a "pure" amplification function. However, the "seed" set of test sequences needs to include the expected responses (i.e., oracle values) as well as the corresponding stimuli (i.e., stimulus/response pairs) to characterise the behaviour of the functional abstraction under test.

The pseudo algorithm of TESTAMP is presented in Figure 14.2. It is similar to TESTGEN's algorithm with the exception that its input is different.

Input: Existing Set of Tests T_{in}, maximum number m of alternative classes to retrieve, b time budget for $MonoGen$, a maximum number of adapters
Output: Amplified test set T_{san} for T_{in}, and n, total number of implementations used to generate tests

1 **Function** `TestAmp`(T_{in}, m, b, a) $\rightarrow T, n$**:**
2 $C_{alt} \leftarrow$ `Harvest`(T_{in}, m)
3 $T \leftarrow T_{in}$
4 **for** $c_i \in C_{alt}$ **do**
5 $T_{c_i} \leftarrow$ `MonoGen`(c_i, b)
6 $T_{adap} \leftarrow$ `Adapt`(c, a, T_{c_i})
7 $T \leftarrow T \cup T_{adap}$
8 **end**
9 $T_{san} \leftarrow$ `Sanitise`(c, T_{adap})
10 $n \leftarrow size(C_{alt})$
11 **return** T_{san}, n

Fig. 14.2.: Pseudo Algorithm of $TestAmp$

The $TestAmp(T_{in}, m, b, a) \rightarrow T, n$ test amplification function takes a test set (i.e., seed), T_{in}, a maximum number of alternative implementations to be retrieved by invoking auxiliary function $Harvest$, m, time budget parameter b for running $MonoGen$ on each available implementation, and the maximum number of adapters, a, to be used to adapt generated tests to the interface of CUT c.

14.3.2 Realisation

Like LASSO TESTGEN, LASSO TESTAMP is realised as a reusable analysis pipeline written in LSL (i.e., script template). The analysis service script in Listing 27 (Appendix) presents one possible realisation of TESTAMP. Here the auxiliary $Harvest$ function (cf. Figure 14.2) is based on TDS.

In this case, the basic pipeline design consists of five LSL actions that apply TESTAMP to a test sequence that verifies the (typical) behaviour of a stack abstraction. Accordingly, the first action of type `Select` retrieves a set of alternative, candidate class implementations based on the interface signature used in the test sequence resembling a stack abstraction. In other words, a preliminary set of classes is harvested using IDCS. Additional filters are defined to exclude undesired classes as for LASSO TESTGEN's example pipeline, on the one hand. But also classes that are classified as "trivial" (i.e., have a cyclomatic complexity less than 10, for demonstration purposes) are rejected as well. Thereafter, the second step executes a filtering action that detects any class duplicates in the set of alternative class implementations and removes them accordingly.

Having obtained an implementationally diverse set of alternative classes with respect to type-2 clones, the next step is to check whether they implement the stack abstraction as specified by the defined sequence sheet. For this, an arena action is defined that acts as a test filtering step to reject all those class implementations that do not exhibit the desired behaviour (i.e., disagree on the stimulus/response pairs). Then, for each matched alternative implementation (i.e., each alternative implementation that is executable and passes the test), the `EvoSuite` action is used to automatically generate additional tests.

The last `Arena` action receives all implementations and their generated test sequences. Since the aim is to return an amplified set of tests and no reference implementation exists, as opposed to TESTGEN, the task is to ensure that the test sequences are compatible to the interface specification of the functional abstraction at hand. As a consequence, the test sequence mining algorithm of the observatorium attempts to adapt the test sequences obtained for the alternative implementations to the target interface. Ultimately, once a set of non-duplicate test sequences has been mined (i.e., "amplified" based on the seed test sequences), a "sanitised" set of tests is returned.

In summary, the novel aspect of TESTAMP is that it only requires a seed test set and no access to the code of a reference implementation of the functional abstraction of interest. It then executes its analysis pipeline to amplify the seed tests. In this way, scenarios can be realised in which users who have written tests before a (partial)

implementation has been developed, can improve the quality of the tests without having access to the SUT (cf. test-first development [33]).

14.3.3 Potential

Similar to the discussion of the potential of TESTGEN in Section 14.2.3, TESTAMP's pipeline can be realised in a variety of ways that are yet to be explored. These include the substitution of actions, alternative approaches for harvesting alternative implementations (e.g., relaxed filtering criteria), different parameter settings for actions, an alternative ordering of actions or extending the pipeline (e.g., test set minimisation).

15 Supporting Experimentation

The second core goal of the proposed observatorium is to support and facilitate experimentation in software engineering. The platform and services required to support controlled experimentation, however, are difficult and expensive [71] to build and apply. In this context, the main advance provided by LASSO is the realisation of a fully-automated software execution service that is accessible via an abstract workflow and data model using a dedicated DSL. This not only opens up new ways of selecting, analysing and comparing software systems systematically, it also frees researchers from many of the tedious and time-consuming tasks involved in traditional observation-based mining and validations tasks, such as curating repositories and writing harnesses to access them (Chapter 7 and Section 13.2). More importantly, LASSO facilitates and automates the sharing of study evaluations and their results, including the test harnesses and results for replication or reuse purposes.

In this chapter, we show how the observatorium proposed in this thesis can be used to better evaluate software engineering tools (cf. Hypothesis 3). We first provide an introduction to software experimentation in general (loosely guided by the work of Wohlin et al. [251]). Then we explain all the basic phases of the experimental process and elaborate on how experimenters can leverage the observatorium presented in this work based on a sample study. Finally, we will demonstrate the capabilities of LASSO with respect to software experimentation based on real studies.

15.1 Experimentation in Software Engineering

The field of software experimentation is generally referred to under the umbrella term of "*Experimental Software Engineering*" (ESE). Historically, software experimentation was not a common activity in software engineering research [214] and the statistical power of software engineering experiments was rather low [73]. Starting in the late 2000s, experimental software engineering gained traction and has matured over recent years. In the same decade, a research community formed under the umbrella term of "*Empirical Software Engineering and Measurement*" (ESEM) [77]. ESE is closely linked with the concept of measurement, since it plays a vital role in (i.e., is a prerequisite for) empirical studies.

The importance of empirical research has significantly grown over the years since almost all premier software engineering conferences and venues (e.g., journals) require authors of papers and articles to conduct empirical evaluations [211]. Studies (i.e., experiments) in software engineering are needed (or useful) for a variety of reasons including the evaluation of software products and processes by engineers in industry and the evaluation of tools and techniques by researchers in academia.

Since the observatorium mainly targets software products (i.e., source code and related artefacts) and software resources in terms of tools and techniques, the content in the remainder of this chapter is tailored to these accordingly. In this context, researchers often ask questions like "has X a significant effect on Y?". In this case X often depicts a tool or technique and Y a performance improvement (e.g., efficiency improvement). Studies of this kind typically employ some sort of "*benchmarking*" where X is compared to one or more baselines. Typically, such studies are performed using *controlled experiments* which are the main type of studies supported by the observatorium. There are other experimental strategies that include case studies, interviews or surveys/questionnaires, but these are usually performed manually by humans.

Experimentation in general builds around three core concepts that are aligned with a set of research questions determined by the scope and goal of the study, usually formulated in terms of hypotheses —

- *Measurement*: defining measurements for the object's attributes under study,
- *Data Collection*: obtaining quantitative/qualitative data from measurements for particular research questions,
- *Data Analysis*: for each research question, analysing collected data using statistical testing.

The measurement of attributes of software systems and tools and techniques falls into three categories. Firstly, a measurement can be either *directly* obtained from the software system or the tool/technique (e.g., number of tests), or *indirectly* using some high-order metric based on one or more attributes (e.g., test coverage measured in terms of mutation score). Secondly, based on the presence of human judgments, measures may be either *objective* (e.g., LOC) or *subjective* (e.g., a questionnaire or estimation provided by humans). Thirdly, measures can be expressed as either *quantitative* (numerical data) or *qualitative* data (e.g., interpretation).

The observatorium mainly operates on quantitative data that is measured and collected automatically in the analysis steps (i.e., actions) specified in LSL pipelines. Whereas direct measures can be obtained through scope-aware measurements (Chapter 5), for instance, indirect measures (i.e., compound metrics) can be formulated

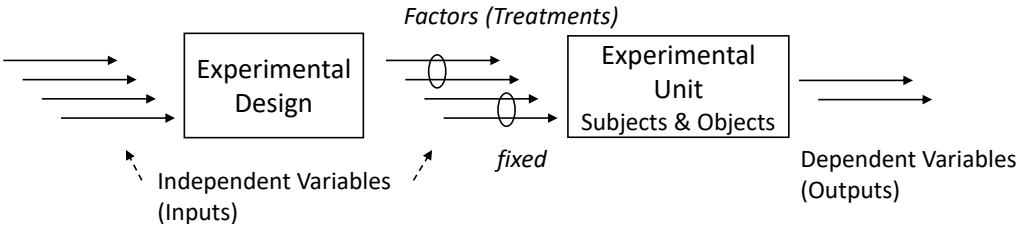


Fig. 15.1.: A Controlled Experiment Based on Wohlin et al. [251]

either (a) using predefined actions (Section 12.5), (b) in LSL actions (Section 13) as part of the LSL study script, or (c) as part of the data-driven analysis in the data analytics layer of the observatorium (Section 11.2). While objective data is automatically obtained in LSL study executions, users have the option to load existing, manually created subjective data into SRMs using LSL actions (e.g., labelling data for classification purposes). Obviously, the gathering of qualitative data (e.g., interviews) cannot be automated per se by the observatorium. The numerical data extracted from it, on the other hand, may be integrated and studied in the data analytics layer.

Data analysis is typically carried out in terms of statistical testing and has the objective of attaining study results with statistical power. This capability is provided by the observatorium's data-driven analysis capability in the data analytics layer that integrates closely with external statistics systems such as large-scale data analytics platforms. More specifically, the SRM-related records (i.e., analysis attributes) measured and stored in the observatorium are represented as data frames that end-users can use to conduct their statistical testing tasks (e.g., first by exploring the measurement data using descriptive statistics).

15.2 Experimental Process in the Observatorium

Studies are typically defined and conducted using an experimental framework (e.g., Basili et al. [29] or Wohlin et al. [251]). However, before we introduce the phases in a controlled experiment, we first introduce the basic terminology motivated by Wohlin et al. [251] to establish a common understanding. Figure 15.1 presents an abstract overview of an experiment including some core notions.

Formally, a (controlled) *experiment* is the process of applying one or more operations to an experimental unit of objects and subjects. The objective is to observe the effect of the operations by measuring them on one or more dependent variables (also called response variables). *Dependent variables* typically depict an attribute or characteristic of interest in the object under study, while *independent variables* are identified by the experimenter with the hope that the dependent variable are

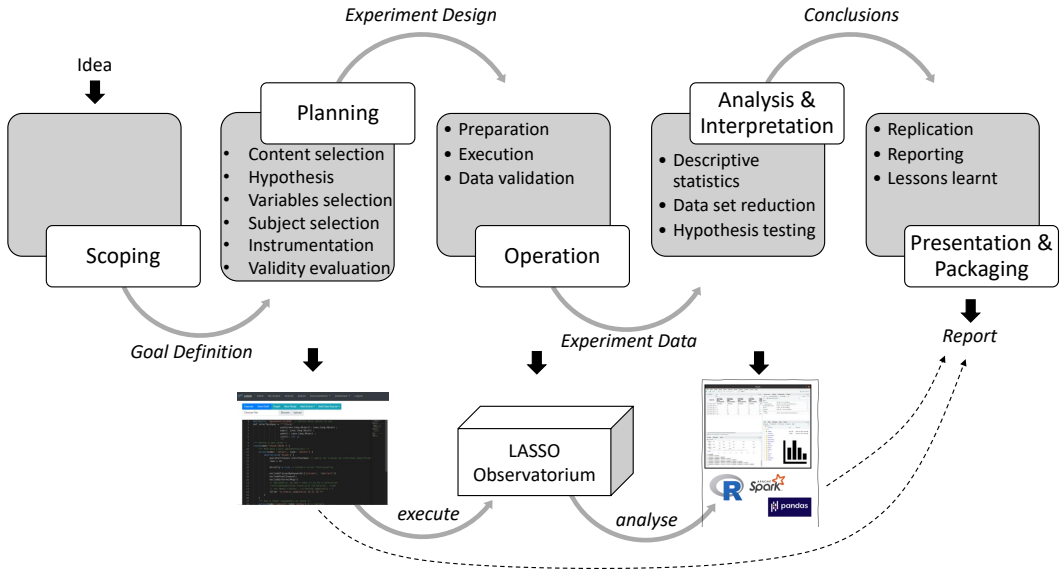


Fig. 15.2.: Conducting Studies in the Observatorium Based on Experimental Process of Wohlin et al. [251]

affected by them. The values of the independent variables identified as part of the experimental design are controlled by the experimenter. Typically, a small set of independent variables (one or more) is set to a certain value (i.e., operations in terms of explainable factors are applied), while the remaining independent variables stay fixed to certain constant values in order to control the observable effect on the dependent variables (e.g., environmental variables). While independent variables present the inputs to the experimental process, dependent variables can be thought of as the outputs.

An *object* in an experiment depicts the entity under study that is subject to operations, whereas *subjects* in the experiment (sometimes called participants) are the entities to which operations are applied. Figure 15.2 provides an overview of the five phases of the experimental process proposed by Wohlin et al. [251] and illustrates how these can be applied to conduct experiments (i.e., LSL studies) in the observatorium presented in this thesis.

The purpose of the *scoping phase* is to focus on the problem of the study and to clearly define the goals. The *planning phase* then fleshes out the experimental design of the study including the formulation of one or more hypotheses about the object's attributes which involves the creation of several research questions and corresponding measurements of the dependent variables.

The study is executed in the *operation phase* that collects the measurement data about the dependent variables and validates them (i.e., post-processing of data). The ("valid") measurement data is then analysed and interpreted in the *analysis*

and interpretation phase involving the characterisation of the data, followed by hypothesis testing to attain statistical power. Finally, in order to communicate the results and make them available to others, a report is written (e.g., paper/article) and (hopefully) a study package for replication is provided as part of the *presentation and package phase*. Note that the experimental process is not intended to be a "waterfall model" in a strict sense. Each phase in the process can be rolled back or carried out iteratively to a certain degree in order to incorporate feedback, with the sole exception of the operation phase which is special in this respect. Depending on the "costs" of the study, the operation phase may be a costly endeavour that cannot simply be stopped and resumed at certain points in time. For example, suppose that an experiment is run on a (super)computing grid that is only bookable for a limited time frame. In this case, the operation phase of the study is rather rigid and resuming it may mean that experimenters have to wait a long time until they can book the computing grid again.

In the following subsections, we explain each phase in greater detail and discuss how LSL studies can be systematically composed and conducted in LASSO's observatorium based on a sample tool study.

15.2.1 Example Tool Study - EvoSuite

To perform a study using LASSO, researchers translate their study design into an executable form using LSL. To exemplify the process, we discuss how the experimental phases are applied to write LSL studies by walking through a realistic evaluation of the AUTG tool EvoSuite (see Section 16.4.2), inspired by tutorials on running experiments with that tool [81, 82]. The aim is to study the effect of search time budgets on the quality of the generated set of test classes. Using the terminology introduced previously, EvoSuite represents the object (i.e., tool) of the study and test quality is the attribute of interest. The search time budget is an important parameter (i.e., independent variable) of EvoSuite's underlying meta-heuristics algorithm and is believed to have an effect on the test quality (i.e., dependent variable) of the tests generated. In other words, intuitively, the hypothesis is that the more time assigned to the test generation process, the higher the quality of the tests generated by EvoSuite.

It is certainly possible (and encouraged) to rephrase the experiment by saying that the object under study is the actual algorithm used by EvoSuite (i.e., the DynaMOSA algorithm), but in this case, this does not change the meaning of the study. Note that in the context of meta-heuristics AUTG tools there are many other interesting experimental designs, all of which do some kind of "benchmarking". In our case, we perform benchmarking by comparing different configurations of Evo-

SUITE with respect to the assigned search time budgets. More generally, parameter tuning of meta-heuristics approaches is a core problem (cf. no-free lunch theorem [252]) that deserves further investigation [12]. Apart from the search time budget, many more parameters can be identified that may serve as independent variables of test quality (e.g., all the remaining parameters of the algorithm used including its multi-objective criteria defined etc.).

The most basic study in search-based software engineering is to demonstrate that the tool/technique at hand is superior to random testing techniques (e.g., to evaluate whether EVOSUITE generates better tests than a random testing tool such as RANDOOP). EVOSUITE is often used as a prototype platform to compare improved or new algorithms. For example, Panichella et al. [187] demonstrated that their DYNAMOSA algorithm performs better than Fraser and Arcuri's whole-suite test generation algorithm [85] using EVOSUITE. In this benchmarking study, the whole-suite test generation algorithm served as the baseline for comparison.

Another approach for further studying EVOSUITE's effectiveness is to run it on subjects (i.e., Java classes) other than the default curated corpus SF110 [83] on which it is often evaluated. This type of study constitutes a *replication* of existing results on a different set of Java classes in order to determine if published results are generalisable. This is especially important to establish whether EVOSUITE's capabilities apply to other sets of Java classes as well (e.g., real-world classes sampled from large repositories such as Maven Central).

15.2.2 Scoping

The first phase in the experimental process is scoping where the problem of the study is identified and formulated in terms of objectives and goals. This activity includes the formulation of one or more (informal) hypotheses. Informally, we already carried out the first activity by explaining the sample study and its goal. In practice, however, researchers are encouraged to use a more systematic approach to define the goal of an experiment such as the goal template framework based on Basili's GQM paradigm [28] (goal-question-metric, see Section 5.3.1). The goal template contains five basic entities which are mapped to our sample study as follows —

- (*Object of Study*) Analyse EVOSUITE and its algorithm DYNAMOSA,
- (*Purpose*) for the purpose of automatic test unit generation,
- (*Quality Focus*) with respect to test quality (effectiveness),
- (*Perspective*) from the point of view of the engineer (i.e., practitioner),
- (*Context*) in the context of Java classes harvested from Maven Central.

The entities of the goal template provide concrete guidance and directions for the next phase, the creation of the experimental design and its corresponding LSL study script in the planning phase.

15.2.3 Planning

As its name implies, the planning phase fleshes out the detailed design of the experiment. The environment and setting in which we conduct the environment is the observatorium and its underlying executable corpus. Since EvoSuite only supports Java classes, LASSO is a suitable experimental environment and realises the "instrumentation" task of the study (i.e., the execution and collection of data).

The planning action is the main experimental phase in which the design of the LSL study is elaborated. Listing 20 provides a possible design of the resulting LSL script for our sample tool study. The study is split into five basic actions, some of which are executed repeatedly based on the given time budget values and the number of repetitions. Each part of the LSL study defined in the given script is defined below based on the common tasks in the planning phase.

Variables Selection and Hypothesis Formulation

Typically, the planning activity includes the formal definition of the hypotheses and the identification of the inputs and the outputs of the envisaged experimental process in terms of independent variables and dependent variables. The selection of variables is a creative task that needs to be done by the experimenter. In our sample study, we determined test quality as the dependent variable and hypothesise that it is affected by the search time budget parameter of EvoSuite's underlying algorithm. Having identified the single independent variable (i.e., search time budget), we need to identify a list of factors that we want to use for its application in EvoSuite. In other words, we need to identify particular values for it. In our case, we want to assess and compare test quality based on two applications of the independent variable, here by choosing two time budgets, 30 and 60 seconds. Other values for time budgets can be directly encoded as parameters provided to corresponding LSL actions. As presented in the list structure in Line 4 of Listing 20, here a global variable is defined in the preamble of the LSL study script that is referenced by the `EvoSuite` action (analysis step) in the remainder of the script (cf. Line 38 and 41). A potential null hypothesis that we want to reject in this case is that the time budgets do not lead to any differences with respect to the test quality of the generated sets on the subjects.

Note that it is important to also identify independent variables other than the search time budget parameter. Ideally, we consider the whole input space of potential

```
dataSource 'mavenCentral2020'

def randomClassesTotal = 20 // number of classes to randomly sample
def repetitions = 10 // number of repetitions
def timeBudgets = [30, 60] // time budgets in seconds

study(name:'EvoSuite_TimeBudgets') {
    action(name: 'selectRandom', type: 'Select') { // random sampling
        abstraction('Random') {
            queryForClasses '*:*', 'concept'
            rows = randomClassesTotal
            random = true
            excludeClassesByKeywords(['private', 'abstract'])
            excludeTestClasses()
            excludeInternalPkgs()
            excludeExceptions()
            // rule out trivial classes
            filter 'methods:[1 TO *]'
            filter 'branches:[10 TO *]'
        }
    }

    action(name: "clones", type: 'Nicad6') { // reject code clones
        cloneType = "type2"
        collapseClones = true

        dependsOn "selectRandom"
        includeAbstractions 'Random'
        profile {
            environment('nicad') {
                image = 'nicad:6.2'
            }
        }
    }

    profile('evosuite') { // execution profile
        scope('class') { type = 'class' }
        environment('java8') {
            image = 'maven:3.5.4-jdk-8'
        }
    }

    for(int repetition = 0; repetition < repetitions; repetition++) { // repeat
        for(int timeBudget : timeBudgets) {
            action(name:"evosuite_${timeBudget}_${repetition}",type:'Evosuite') { // run
            ↪ EvoSuite
                version = '1.1.0'
                searchBudget = timeBudget
                // other parameters: criteria, algorithm etc.
                dependsOn 'clones'
                includeAbstractions 'Random'
                profile('evosuite')
            }
            action(name:"pitest_${timeBudget}_${repetition}",type:'Pitest') { // measure MS
                dependsOn "evosuite_${timeBudget}_${repetition}"
                includeAbstractions 'Random'
                profile('evosuite')
            }
            action(name:"jacoco_${timeBudget}_${repetition}",type:'JaCoCo') { // measure BC
                minimumTestCoverage = 0d

                dependsOn "evosuite_${timeBudget}_${repetition}"
                includeAbstractions 'Random'
                profile('evosuite')
            }
        }
    }
}
```

List. 20: LSL Script of Sample EvoSuite Tool Study

parameters available to EvoSuite and attempt to control them by fixing their values, thereby keeping them constant while varying the search time budget. Action `EvoSuite` in the given LSL script provides default values for the remaining major parameters based on EvoSuite's default parameter choices. These are not explicitly provided by the action.

In practice, however, there are many more independent variables that may affect the dependent variable such as environmental factors. These need to be controlled as well (e.g., software and hardware configuration). A core capability of the observatorium is to allow execution profiles to be defined by the experimenter to control these factors as illustrated by the profile block in Line 31 (cf. Listing 20). The provision of the profile ensures that all executions of the EvoSuite tool are performed in the same execution environment. An advanced configuration option even allows the hardware of the machines in the compute cluster of LASSO to be fixed. So in practice, a set of homogenous computing machines is used to conduct the experiment in order to prevent any measurement bias induced through varying computing power. In the case of EvoSuite, this is of particular importance since machines of varying computing power may consume the search time budgets in different ways (i.e., more powerful machines apply more computing power in the same time frame).

Selecting Subjects

An important task in the design of the study is to select subjects on which EvoSuite's test quality effectiveness is assessed. In our study these represent real-world Java classes harvested from Maven Central (see Section 12.4.1). As well as providing this particular data set, LASSO also offers existing software engineering corpora frequently used for benchmarking purposes including SF110 (overview in Section 12.4.2). However, in order to demonstrate the versatility of the executable corpus provided by LASSO, and to select from a much larger number of practical Java classes, we choose Maven Central in this example.

The EvoSuite test generation process is costly. In order to scale to many Java classes, we need to choose an appropriate selection strategy for the subjects (i.e., selecting a subset of classes from the Maven Central data set). To increase statistical representativeness, the randomisation of subjects is key to make claims about the effect of search time budgets on test set quality. Another key ingredient to obtain generalisable results is population size with respect to the number of subjects (i.e., classes) sampled.

Sampling Strategies As explained in the context of LASSO CURATE, experimenters can choose from several available sampling strategies (cf. overview in [20]) to select executable Java classes from the executable corpus. In our case, the sampling strategy is behaviour-agnostic, since our study goal is to sample classes with any behaviour. In order to select executable Java classes, we use the `Select` action provided by LASSO that is configured for random sampling (i.e., specified via the "random" parameter) coupled with the "rows" parameter that determines the total number of Java classes to return.

Note that with respect to population size, LSL scripts can be executed in batches and the collected results can be joined afterwards. This allows experimenters to attain intermediate results that provide valuable feedback about the operation phase. A discussion of possible divide-and-conquer strategies to refine the experimental design is provided in the subsequent sections.

Multi-Criteria Constraints The selection of executable classes (i.e., subjects) can be further constrained at a fine-grained level by defining multiple criteria based on the class properties stored in the corpus to reject classes that are not of interest for the study or object at hand. In our study, we filter out all Java classes that are either incompatible, not useful or too trivial for EVOSUITE. This is often necessary, since test classes cannot be created for Java classes that are invisible (i.e., Java access modifiers), are test classes themselves, or represent classes from internal facilities such as EVOSUITE's internal classes. Moreover, trivial classes such as plain old Java objects (so-called POJO's) that only declare setter and getter methods, but no non-trivial behaviour, are rejected as well, since they do not add much information to the study results. We conjecture that in this case, the challenge of generating tests is trivial, since only a simple set of paths need to be explored by EVOSUITE. The multiple criteria used in this example are based on indications provided by the static code measures stored in the executable corpus (i.e., the number of methods must be at least 1 and the number of branches must be greater 10).

Diversity In many cases, the subjects selected for a study need to exhibit certain characteristics to not interfere with the experimental design. A core property that is desirable in formal experimentation in software engineering is *diversity* [179]. Diversity can occur in a variety of ways, but commonly it is defined based on the notion of software redundancy. A key issue that needs to be addressed to achieve a fully random selection process, with no manual intervention whatsoever, is to automatically ensure that results are not biased by a significant proportion of code duplication. The observatorium provides a clear definition of redundancy in terms

of simple and heteromorphic redundancy (Section 14.1) that serves as a starting point to ease the decision-making process of experimenters.

The study under consideration is behaviour-agnostic, so it is sufficient to attain diversity by randomly sampling over a large search space of Java classes (i.e., Maven Central data set) as well as by employing an additional filtering step based on the idea of rejecting any code duplicates, here using the code clone detection tool NICAD. The action block defined in Line 20 of Listing 20 attempts to detect and reject any classes returned by the `Select` action that are considered up to type-2 clones.

Testability Another important property of the subject classes is that they must be testable. In other words, the classes processed by the LSL pipeline need to be executable by EVOSUITE. Note that by using LASSO, the experimenter is freed from the work often involved in making returned classes testable. The observatorium automatically ensures that each class passed to the action is executable. Any class that fails to satisfy this property is "marked" and dropped from the pipeline. The reporting capability of LASSO keeps track of any failed execution attempts.

Instrumentation and Measurement

Having selected a population of subjects that exhibit certain characteristics, it is now time for the instrumentation phase of the study. This task includes the identification and preparation of the object of the study. In our case, we need to pick a certain version of the EVOSUITE tool, and we need to define its controlled execution environment in which one or more measurements can be obtained.

The observatorium facilitates this task by either choosing a predefined action that already integrates the object (i.e., EVOSUITE), or the experimenter may decide to integrate an existing tool/technique using LASSO's extensible Actions API (Section 12.5.1). As discussed in the previous chapters, LSL actions can be configured at a fine-grained level. In terms of the EVOSUITE action that is already predefined, the actual tool version can be configured and all its major parameter settings as well as its execution environment can be defined in terms of execution profiles (see `profile` block in the script). The configurability of actions and profiles allow environmental factors (i.e., independent variables) to be fixed in order to control the effect of the varied search time budget on test quality.

Measurement Procedures The next step is to develop one or more measurement procedures that measure data related to test quality (dependent variable – how do we measure effectiveness in terms of test quality?). In order to define measurement procedures, researchers are advised to apply the GQM paradigm in order to break

down the study goal into research questions, each with identified metrics related to the goal that can be measured.

There are common test coverage criteria candidates that can be measured to indicate test quality. Fraser and Arcuri [83] typically focus on branch- and mutation coverage criteria, whereas Panichella et al. also included statement (line) coverage to assess their proposed DynaMOSA algorithm [187]. Accordingly, we have selected branch coverage measurement using JaCoCo as well as mutation coverage measurement using PIT. Note that EvoSuite provides a coverage measurement harness on its own, so data reported by EvoSuite is collected by LASSO as well. One of the reasons for including additional test coverage measurements, however, is to enable the validation of measurements and to identify potential measurement errors. As a consequence, we can define two research questions based on the measurement of mutation score and branch coverage as follows —

1. How does a time budget of 30 seconds perform compared to a time budget of 60 seconds with respect to mutation score?

2. How does a time budget of 30 seconds perform compared to a time budget of 60 seconds with respect to branch coverage?

As can be seen in Listing 20, the measurement procedures are directly encoded into the LSL pipeline by adding corresponding actions of type `Pitest` (for PIT) and `JaCoCo`. The measurements are performed for each instance of EvoSuite based on the given time budget used (see inner for-loop). Note that all three actions use the same execution profile that defines the same, controlled execution environment (i.e., Java 8 and a common container image, see Section 12.3) in order to prevent measurement bias and to ensure the comparability of the data collected.

After the definition of controllable execution environments, the `profile` block also allows experimenters to define a certain measurement scope that determines the code elements to be analysed (Section 5.2). The scope definitions affect both EvoSuite's analysis step and the measurement steps of JaCoCo and PIT. By default, EvoSuite identifies all "visible" methods of a class for which tests are generated. That is why the scope is set to "class". Alternatively, the scope can be further restricted by excluding certain methods (if necessary) via the provision of a black or white list.

In conclusion, LASSO frees experimenters from having to set up the aforementioned measurement tools in a way that scales with the number of classes (i.e., subjects), since these are already integrated into the observatorium's systematic measurement architecture.

Threats to Validity Some experimental designs require further precautions (i.e., against threats of validity) in order to obtain valid results with statistical power. In fact, our study object uses a randomised algorithm which can yield different results from different executions on the same CUT under the same circumstances. In order to assess the performance of such algorithms in EvoSuite, Arcuri and Briand propose that test generation should be performed multiple times for a particular class (e.g., 10 or 30 repetitions) in order to attain sufficient statistical power [10].

To mitigate the threat of invalid data, repetitions can be directly realised in LSL pipelines by using simple iteration. The example study script in Listing 20 uses a classic "for" loop to perform repetitions of certain actions (cf. Line 37). This ensures each value for the independent variable (i.e., search budget) is measured for mutation- and branch coverage.

Note that the choice of the number of repetitions is a research question in its own right. In general, it needs to be balanced with the population size (i.e., number of subjects in the study) along with the available study budget (i.e., available machines, time and human effort). The number of executions in this study can be approximated by $\#SearchBudgets \times \#Classes \times \#Repetitions$, and can obviously grow quickly, like the number of coverage measurements.

Script Design The final LSL pipeline script consists of five basic actions that realise all our experimental design choices in 60 lines of code. Moreover, almost all design choices are realised in a declarative way apart from two nested loop structures. The final structure of the script not only demonstrates the effectiveness of LSL pipelines and the expressiveness of LSL, it also provides a template for future modifications in case the experimental design needs to be refined based on intermediate feedback from running the script. The preamble in the pipeline script, for instance, defines basic properties of our study that we can simply adjust by changing the values. For example, we can extend the study and include additional variations of the independent variable (i.e., search time budget) by simply extending the list structure.

A practical example is to conduct pilot studies (i.e., quick mini studies) based on a smaller size of population and a smaller number of repetitions in order to obtain sensible defaults for independent parameters or more generally, to check the experimental design for basic feasibility in terms of exhibited run-time behaviour and potential programming mistakes etc.

15.2.4 Operation

The next phase involves the execution of the actual study. The operation phase consists of three basic steps: (1) preparation of the class subjects and the execution

environment, (2) their actual execution, and (3) validation of the data collected from the execution (i.e., the measurements). With respect to our study script, both the preparation of classes, their execution via EvoSuite and the measurement and collection of evaluation criteria for the generated test sets is fully automated. The last step, however, may require intervention by the experimenters, either by using LSL commands or performing validation in the data-driven analytics layer of the observatorium. Often this step is carried out as part of a "data cleansing" step in the analysis phase. The goal is to spot any missing or incorrect measurements and to identify potential design and measurement flaws in the experimental design.

Fail Safety and Robustness Arguably, the higher the number of executions, the higher the likelihood that something will fail in the experiment. An important design principle built into the observatorium, therefore, is the property of *fail-safety*. A general-purpose script such as a Python script (e.g., as used in [82]) may fail at any point during execution, since it may not have been optimised for execution robustness (e.g., unforeseen failures can happen at any point in time). Experimenters can control the behaviour of the observatorium's fail-safety by providing the parameter `dropFailed = true` if failed classes in an action should be removed automatically or if they should be kept. In our case, we keep all classes, even the failed ones, since several EvoSuite executions may exhibit different failing behaviour (based on our experience). Failed instances of classes can be later analysed and possibly rejected as part of the next phase, the analysis and interpretation of the collected data. Depending on the action under consideration, LASSO stores potential error messages from integrated tools as well. In the case of EvoSuite, these are parsed from the test generation logs. This increases the transparency of the study operation phase and provides valuable feedback for experimenters to facilitate decision-making (e.g., refining study and measurement design etc.).

15.2.5 Analysis and Interpretation

Once the LSL study script has been executed in the observatorium, the obtained measures (i.e., SRM records) are automatically collected and stored in LASSO's transactional database from which it can be obtained for data-driven analysis.

The analysis and interpretation phase first starts with an informal analysis of the measures collected by the given LSL actions in order to understand them. For this purpose descriptive statistics are used to characterise the data (i.e., test coverage measurements in terms of mutation score and branch coverage). Based on a first characterisation, which may include some sort of data visualisation based on plotting (e.g., bar and whisker plots that summarise descriptive statistics such as quartiles,

```r
library(readr) # read CSV as data frame, https://readr.tidyverse.org/
library(dplyr) # grammar for data manipulation, https://dplyr.tidyverse.org/

# CSV export or use RJDBC - library(RJDBC)
evosuite_records <- read(...)
# get mutation score / branch coverage
mutation_scores <- evosuite_records %>% filter(TYPE = "PITEST.MUTATION_SCORE")
branch_coverage <- evosuite_records %>% filter(TYPE = "JACOCO.BRANCHES")
mutation_scores
# descriptive statistics / create box plot (action has unique name)
summary(mutation_scores %>% filter(starts_with(ACTION, "pitestEvoTime_30")) %>%
↪  group_by(ACTION))
boxplot(mutation_scores %>% filter(starts_with(ACTION, "pitestEvoTime_60"))
↪  %>%group_by(ACTION))
...
# data cleansing (data set reduction)
mutation_scores %>% filter(is.na(VALUE)) # data points not available
...
# statistical testing: comparing time budget 30 (a) to 60 (b)
wilcoxon.test(a,b) # library(stats)
library(effsize) # Vargha and Delaney A effect size measure
VD.A(a,b)
```

List. 21: Excerpt of R Script for Analysis and Interpretation of Measures Collected for Sample Study

standard deviation and outliers), experimenters may need to manipulate the data and drop invalid data points. This task is also referred to as data set reduction. Once experimenters have obtained a good understanding of the data and have cleaned it, they can apply statistical testing in order to confirm or reject the hypotheses formulated in the planning phase of the study.

Note that the analysis and interpretation activity is usually performed in an interactive way and relies on the knowledge and creativity of the experimenters. Once experimenters have gained enough confidence in the quality of the underlying data, they assemble scripts that automate the generation of results.

As explained in the previous chapters, LASSO provides a set of connectors for popular data analytic platforms. Listing 21 provides a sample R program that illustrates the analysis phase for the sample study of EvoSuite. First, the observational records of the observatorium are read in using an appropriate connector. Based on the raw data frame of SRM records, we can select the records corresponding to the mutation score and branch coverage measurements. Thereafter, we characterise each coverage measurement by printing its summary statistics to the console and by visualising them using a box and whisker plot. Note that we group the data based on the chosen time budgets. The next step involves the cleaning of the measures by identifying invalid values such as missing measurements.

Once clean data frames have been obtained, we continue with the testing of our hypothesis that larger time budgets increase the effectiveness of EvoSuite in terms of the test quality of the generated tests. In our particular example, we guide our

statistical testing using the approach proposed by Arcuri and Briand [10]. In this case, the non-parametric Mann-Whitney U test (Wilcoxon test) [167, 53] and Vargha and Delaney [244] effect size measurements are applied to confirm or reject the null hypothesis.

15.2.6 Presentation and Package

Finally, the last phase of the experimental process involves the presentation of the finding from the former phase and the packaging of the study results. Researchers typically write reports and publish them as papers or articles. These contain a critical, scientific discussion of the main findings (including a list of threats to validity [211]), discoveries and a presentation of the lessons learnt from the study.

Replication

Many conferences and journals currently expect researchers to provide a package that documents how results and findings were obtained, often including the tools and analysis scripts used in order to enable the replication of the results and findings.

Sharability Using the observatorium, the creation of a replication package can be automated to a high degree. Once an experimenter submits an LSL study script, the observatorium creates a workspace environment in which the script and its execution-related resources are stored (including execution data and records stored in the database). Together with the statistical analysis scripts (e.g., the R script above), the workspace can be shared with other researchers such as reviewers. These can inspect the study design by browsing the LSL study script and the results obtained.

Flexibility, Variability and Reuse Replications of a study are not always meant to be one-to-one replications, but they may also include variations and extensions of existing study designs. Variations of a study design are made possible through LSL scripts, since we can change certain (independent) variables of the existing study design. In our case, if we choose to increase the number of repetitions, for example, the only change necessary to the study script is to replace the value 10 with the value 30. As well as variations of study design, future researchers may opt to extend the study design to also explore more factors related the independent variable – time budget. In order to realise this change, it is only necessary to extend the list of time budgets given in the preamble of the script with a list of additional time budget values. The general structure of the experiment stays intact. More advanced

examples of extensions of the study design involve the inclusion of more objects or additional tools. Imagine we want to compare EvoSuite using certain time budgets with the random AUTG tool Randoop, for example. In this case, we need to change the pipeline structure by adding another action that executes Randoop on the classes under study.

Reusing Subjects The class subjects that were automatically curated using the selection criteria defined by the experimenter may be reused in other studies. Similarly, another form of variation of an existing study design using its LSL pipeline is to change the underlying data source and set it to a different one or additional ones. For example, we used the data source Maven Central that contains a large set of classes. In order to maintain comparability to results obtained by past studies that were not conducted in the observatorium, the experimenter can reuse existing integrated corpora that are available in the executable corpus.

Integrating Past Studies and Designs Finally, the existing study design can also be changed to study the default choice for time budgets which is 120 seconds. In this case, we are able to replicate studies in the observatorium that were conducted using different instrumentation and a different corpora (e.g., SF110 [83]). The ability to support the replication of existing studies in the observatorium demonstrates its power, since it facilitates the sharing and validation of past results.

15.3 A Practical Study - Diversity-Driven Test Generation

In order to further demonstrate the utility of the LASSO platform for software experimentation, this section explains how it was used to conduct a large-scale study of LASSO TestGen (cf. [138]), the diversity-driven test generation service built on top of the LASSO platform (see Section 14.2).

The applied study design requires a complex technical setup that is challenging to get right. Experimenters that have to use the traditional "toolboxes" (e.g., custom shell scripts or other custom solutions) need to spend a lot of time to set up the study design correctly, to automate it, and most importantly to make it robust enough to work on a larger scale with possibly many unknowns (e.g., foreign software systems harvested from large repositories). Using LASSO's LSL scripts, we demonstrate that such a complex setup can be realised using LSL pipelines that offer a high degree of automation. These pipelines result in scripts of manageable size that can (a) be understood by third-parties, (b) used for replication purposes, and (c) realise a study design that allows for rigorous statistical testing of the obtained data.

First, we explain how we translated the benchmarking study design planned for the assessment of LASSO TESTGEN into executable study pipelines written in LSL. Second, we present the main insights from the obtained data. Further details about the study are provided in [138]. Since we already discussed all the phases of experimentation and how LASSO supports them, this section focuses on the core steps taken to obtain generalisable results of statistical significance.

15.3.1 Scoping and Planning

The basic goal of our study was to assess whether exploiting software system diversity in large software repositories is beneficial when generating tests using the AUTG tool EVOSUITE (referred to as $MonoGen$, for mono-implementation based, see Section 14.2). The scope of the study was the evaluation of the practical feasibility of diversity-driven test generation using our prototype implementation LASSO TESTGEN. Using the GQM goal template introduced in Section 15.2.2, the study goal can be formulated as follows —

- (*Object of Study*) Analyse LASSO TESTGEN and its underlying algorithm,
- (*Purpose*) for the purpose of automatic test unit generation,
- (*Quality Focus*) with respect to test quality (effectiveness),
- (*Perspective*) from the point of view of practitioners,
- (*Context*) in the context of real-world Java classes harvested from Maven Central.

In order to establish a real-world setting, therefore, we evaluated LASSO TESTGEN using Java class subjects sampled from the Maven Central corpus. Since LASSO TESTGEN relies on the diversity found in real-world repositories, we believe that class samples from such a real-world corpus provide a good indication of the results practitioners may experience in real-world software projects.

The study design basically resembles a classic benchmarking study as used in our example tool study. We benchmarked the study object LASSO TESTGEN against two variants of $MonoGen$ (i.e., configurations of EVOSUITE) on class subjects randomly sampled from Maven Central running under the same resources and overall time budgets (i.e., factors/treatments) with respect to their achieved test set quality (i.e., dependent variable). More specifically, the benchmark compared three concrete study objects (cf. Section 14.2.1) —

- $TestGen_{2n}$: We set m of function $TestGen(c, m, b, a)$ to allow up to 15 alternative implementations harvested from Maven Central,

- $MonoGen_2$: We set time budget b of function $MonoGen(c, b)$ to the default time budget setting of EvoSuite in order to provide a baseline (i.e., 2 minutes),
- $MonoGen_{2n}$: We set b of $MonoGen(c, b)$ to $2 * n$ to use the same budget available to $TestGen_{2n}$, where n is the number of class implementations processed by $TestGen$ (i.e., returned as the output n), with a maximum value of 15+1 (i.e., the CUT plus the maximum number of alternative implementations, i.e., $m = 15$ for $TestGen(c, m, b, a)$).

The dependent variable, test set quality, that is compared among the objects, is measured based on the most widely used metrics in the AUTG literature, strong mutation score (MS) and branch coverage (BC).

Since we compared two pairs of study objects, $TestGen_{2n}$ to $MonoGen_{2n}$ and $TestGen_{2n}$ to $MonoGen_2$, and we measured performance (i.e., test set quality) in terms of two metrics, we formulated two core research questions, each of which is symmetrically divided into two subquestions —

RQ1 a) How does $TestGen_{2n}$ perform compared to $MonoGen_{2n}$ on mutation score?

RQ1 b) How does $TestGen_{2n}$ perform compared to $MonoGen_2$ on mutation score?

RQ2 a) How does $TestGen_{2n}$ perform compared to $MonoGen_{2n}$ on branch coverage?

RQ2 b) How does $TestGen_{2n}$ perform compared to $MonoGen_2$ on branch coverage?

Operation - Study Pipeline in LSL

In order to translate the aforementioned study design into LSL, we developed a study pipeline for the collection of test quality measurements from the three study objects. The statistical analysis was performed using the external data analytics tool, R, from which we directly accessed all the observational SRM records collected by LASSO.

For the sake of improved readability, we restructured and divided the original study pipeline into two separate LSL scripts that can be executed in a consecutive manner. While Listing 28 presents the study pipeline of the two study objects $TestGen_{2n}$ and $MonoGen_2$, Listing 29 presents the study pipeline of the remaining study object $MonoGen_{2n}$. The latter pipeline depends on the data obtained from the former pipeline. This structural decision demonstrates LASSO's ability to link and access past data in LSL scripts (Section 10.2.3).

While it is certainly possible to combine both pipelines into a single pipeline, splitting them into two scripts improves readability and allows the execution of certain aspects to be planned and delayed to a later point in time in order to sort

out potential issues or to further improve the pipeline (e.g., efficiency). This allows users of LSL to obtain faster feedback in a fail-fast manner in order to speed up the evolution of their study design (i.e., they incorporate improvements based on feedback cycles). Since our case study design contains three objects that consume lots of time, the available study time budget can be better planned and allocated. Note that the study pipelines presented directly encode LASSO TESTGEN in their design (i.e., incorporate the analysis service and its actions provided in Listing 26). However, as explained before, we may opt to "outsource" LASSO TESTGEN into a dedicated LSL script as demonstrated in Listing 26, and link to its results within another pipeline.

Next, we provide a breakdown of the actual study design realised in LSL and its assumptions in terms of actual pipeline steps (i.e., actions).

Sampling of Class Subjects To begin with, we randomly sampled class subjects that served as the CUTs for the AUTG approaches under comparison from the large search space of Maven Central using the select action named `selectRandom` in Listing 28. The strategy of random sampling from a large search space (cf. probability sampling in software engineering [20]) is chosen as a suitable strategy to increase the generalisability of the study results, since studies with a small population can be limited and thus less generalisable.

For space reasons, the full sampling criteria are omitted in the listing provided. In a nutshell they were designed to only retrieve classes that were accessible (i.e., visible), had non-trivial functionality (i.e., at least one defined method and a cyclomatic complexity of at least 10), were production classes (i.e., not test classes) and were not part of the JDK or analysis services (i.e., EVOSUITE and LASSO). Since LASSO's current prototype adaptation service is limited to Java compatible types, classes that used custom types were not considered.

Harvesting Alternative Implementations The second action in Listing 28 realises the $Harvest$ function of TESTGEN (cf. Algorithm 14.1) by running an IDCS for each reference class (i.e., CUT) that was previously randomly sampled. In other words, for each randomly sampled CUT, up to 15 alternative class implementations were retrieved. Note that sometimes IDCS is not able to return the maximum number of alternative classes simply because an insufficient number of classes with similar signatures do not exist in Maven Central.

Increasing System Diversity Once a set of alternative classes has been retrieved for each randomly sampled CUT, the third action in Listing 28 employs a code duplication filter in order to increase their diversity by dropping any class duplicates

up to type-2 code clones. Since class duplicates of the CUT contain the same code units, they resemble simply another run of $MonoGen$ on the CUT and thus introduce a bias to our results. They are therefore viewed as a threat to our study design and results.

Statistical Significance The $MonoGen$ function is realised using EvoSuite. In order to evaluate the study objects (i.e., $TestGen$ and $MonoGen$s), the for-loop in Listing 28 and Listing 29 realise the well-known guidelines of Arcuri and Briand [10] for evaluating inherently randomised algorithms by repeating (test generation) runs multiple times in order to obtain "stable" data points (recall the sample tool study in Section 15.2.1). In order to maximise the number of CUTs analysed in the study, we decided to set the number of repetitions (i.e., for-loop iterations) to 10 which is the minimum number of repetitions recommended by Arcuri and Briand and the number of repetitions used in [187]. All actions nested in the for-loops in Listing 28 and Listing 29 were therefore executed 10 times, thereby ensuring that tests and test quality measures (i.e., BC and MS) were obtained 10 times for each study object, for each randomly sampled CUT.

Test Generation and Measurement of Test Set Quality Each iteration in the for-loop executes two recurring tasks: (1) the test generation using the corresponding study object, and (2) the measurement of the quality of the test sets returned. The first three actions in Listing 28 start with the generation of tests for each reference CUT based on $MonoGen_2$ using EvoSuite. The results are then passed to the `JaCoCo` and `Pitest` actions in order to measure BC and strong MS for the previously generated set of tests. Thereafter, for each reference CUT, tests are generated for all their available alternative class implementations. LASSO TestGen is executed through the arena action which takes in the merged set of classes and their sets of generated tests (including the reference CUTs and their alternative classes). The plain action that follows merges all reference CUTs into one container of systems for the sake of efficient processing by actions for measuring BC and strong MS for the union of generated tests for each reference CUT.

Similarly, the LSL pipeline in Listing 29 involves the same tasks of test generation and measurement of the test sets returned, but for $MonoGen_{2n}$. First, EvoSuite is executed using the same time budget that was used for $TestGen_{2n}$, followed by the measurement of BC and strong MS. Note that the EvoSuite action nested in the for-loop accesses the classes used by the script in Listing 28 for $MonoGen_2$ and $TestGen_{2n}$ via the DSL keyword `dependsOn` (i.e., using LSL's URI scheme presented in Section 10.2.3).

Robustness An important aspect in the task of translating a desired study design into an executable analysis pipeline is to reach a certain level of "stability". As can be seen by the study design, the analysis pipeline is complex and uses a myriad of configuration parameters that are preset to maintain a controllable study design. Due to the (technical) nature of the classes that are retrieved from real-world repositories, study pipelines need to be robust enough to identify invalid data points. Missing or erroneous data points need to be handled with particular care. The goal, therefore, is to detect any measurement bias to the study results as soon as possible in the operation phase (either directly in the pipeline or later in the statistical testing).

In addition to the automatic failure handling capability and the feedback provided by the LASSO platform, the study scripts encode a variety of mechanisms to make the study operation and the obtainment of results more robust, and thus the pipeline more efficient. If tests cannot be generated for a randomly sampled CUT using EvoSuite, for example, that CUT is dropped from the pipeline so that additional actions do not need to be executed to save time. The same strategy applies to the processing of alternative classes. If for any reason (e.g., technical limitation like unresolvable class dependencies), the test generation or measurement on a class fails, it is dropped from the pipeline to speed up processing.

Transparency Readers of LSL scripts are given a lot of details that directly relate to the assumptions made in the study design. More importantly, the majority of the parameters controlled in the study are explicitly specified[1]. This not only facilitates flexible adjustment of studies (e.g., conducting mini studies to explore the final study design), but also enables the "exact" replication of the study by simply re-running the scripts. For example, the study scripts can simply be modified to run the study in different execution environments (e.g., Java 11 instead of Java 8) by changing the profile declarations.

15.3.2 Analysis and Interpretation

The study pipelines were run as manageable batches until they ran out of time. The batching strategy (i.e., running the study pipelines several times) allowed complete results for a smaller set of randomly sampled CUTs to be obtained. This allowed us to continuously monitor the pipeline for valid results. Afterwards, the results of multiple executions were simply merged. We therefore obtained 120 randomly sampled CUTs from Maven Central for which we attempted 10 repetitions for each study object under comparison. Considering total execution times, each of the three

[1] Note that LASSO actions typically set sensitive defaults for many parameters, so users may decide to leave them to their default setting.

Tab. 15.1.: Descriptive Statistics - Overview (120 Randomly Sampled Classes from Maven Central) [138]

	mean	sd	min	max	se	q.25	median	q.75
Randomly Sampled Classes								
# Methods	9.53	6.91	2.00	36.00	0.63	4.00	7.50	13.00
Cyclomatic complexity	23.91	15.50	7.00	89.00	1.42	13.00	19.00	28.00
# Branches	28.39	22.26	10.00	146.00	2.03	15.50	22.00	32.00
# Lines	48.49	38.39	6.00	204.00	3.50	25.00	37.00	56.25
# Mutants generated	34.92	30.24	1.00	200.00	2.76	18.00	26.00	40.00
TestGen								
# Clones detected	7.94	8.29	2.00	46.00	0.93	3.00	4.00	9.00
# Non-clone implementations n	12.96	5.91	0.00	16.00	0.17	14.00	16.00	16.00
# Non-redundant tests harvested	67.93	113.17	0.00	813.00	3.27	7.00	21.00	79.25
Tests Generated								
# Tests $MonoGen_2$	18.41	19.05	0.00	148.00	0.55	6.00	14.00	24.25
# Tests $TestGen_{2n}$	78.35	121.78	0.00	843.00	3.52	10.00	27.50	96.00
# Tests $MonoGen_{2n}$	19.02	21.80	0.00	207.00	0.63	6.00	14.00	25.00

study objects was executed 1200 times in total, resulting in the following execution times for each AUTG approach —

- $MonoGen_2$: $1200 * 2$ minutes,
- $TestGen_{2n}$: $1200 * n * 2$ minutes,
- $MonoGen_{2n}$: $1200 * n * 2$ minutes where n is up to 16 (15 alternative classes plus CUT).

This demonstrates LASSO's ability to facilitate long-lasting, complex studies. Using the batching strategy mentioned above, intermediate results can be obtained more quickly to double-check for issues. Afterwards, the results of substudies can then be simply aggregated. Next, we summarise the results of the analysis of SRMs in the data analytics layer of LASSO using R (similar to the script in Listing 21).

The first step in the analysis and interpretation phase is to describe the data obtained using descriptive statistics. Table 15.1 provides a summary of the basic statistics for the 120 sampled classes from Maven Central. The first part of the table describes their size-based metrics measured using JaCoCo based on the class scope of depth 0 (see Section 5.2). Moreover, it shows also the number of mutants generated using PIT. The second part of Table 15.1 provides statistics on the number of non-clone implementations available to $TestGen_{2n}$, the tests harvested from them as well as the number of class clones rejected, while the final part of Table 15.1 presents statistics about the number of tests generated by each approach.

Findings

Feasibility The analysis and interpretation of the data obtained for $TestGen_{2n}$ suggests that LASSO's Maven Central corpus exhibits a significant level of system diversity that makes $TestGen$ a feasible approach for the diversity-driven generation of tests. Furthermore, it demonstrates that LASSO's adaptation capabilities for adapting tests harvested from alternative classes to the CUT are also effective. $TestGen_{2n}$ achieved a success rate of 83.8% – that is, where it obtained at least one alternative, non-redundant class for the randomly sampled CUTs. Furthermore, it obtained an average of 12.96 non-clone classes to serve as alternative implementations for test generation from which it managed to mine an average of 67.93 non-redundant tests in addition to the tests generated for the CUT alone.

Level of Diversity NICAD identified an average of 7.94 type-2 clones for each alternative class retrieved by the IDCS. Since the diversity found in a set of alternative class implementations for a randomly sampled CUT has a great impact on the effectiveness of $TestGen$, we further analysed the level of diversity found in the sets of non-clone classes for each randomly sampled class. Using the rich set of properties provided by LASSO's underlying corpus as well as its scope-aware measurement technology, we characterised how "different" the harvested class implementations were.

To this end, we picked two groups of properties, one which provides indicators of the origin and purpose of classes (based on the idea of N-version programming [16] that different software developers write different implementations of functional abstractions), and one which provides key indicators of structural complexity that were obtained using JACOCO based on the shallow, source-based scope $scope(class)$.

For each category, we picked three properties. A visualisation of the spread of property values for the 120 randomly sampled classes and their functional abstractions studied is provided in Figure 15.3. While the top three histograms show the three properties of the first category, the three properties of the second category are shown at the bottom. Each plot shows the frequency distribution of the numbers of harvested implementations with distinct properties for each of the 120 randomly sampled classes.

The analysis of the histograms suggest that there is a significant level of diversity present for the majority of our randomly sampled classes and their harvested alternative implementations, since the frequencies of all six histograms mainly peak in the second half of the plots.

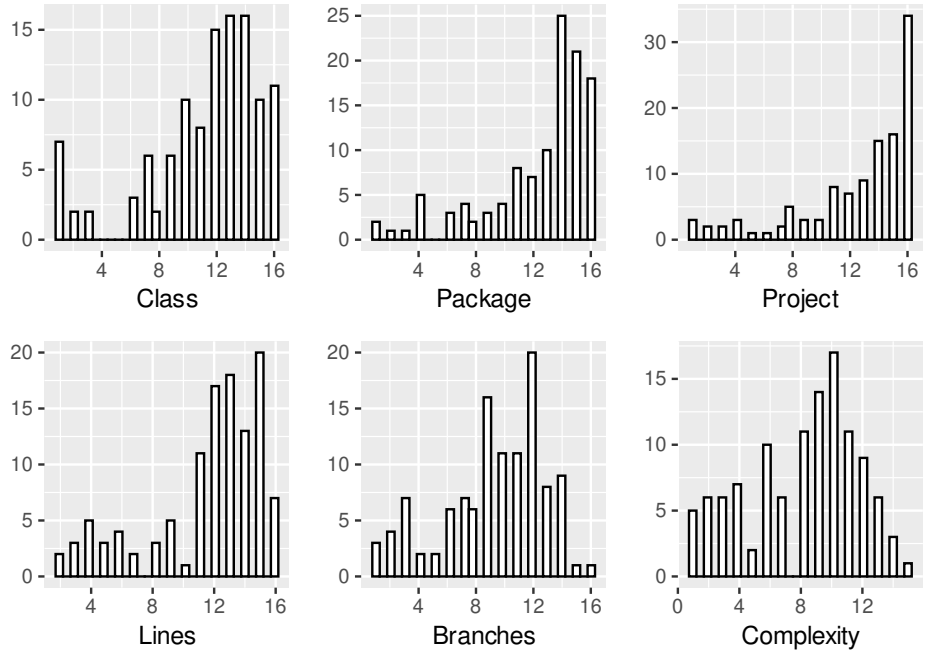


Fig. 15.3.: Diversity Indicators - Histogram of Frequency Distributions of Key Properties [138]

Rigorous, Statistical Testing In order to answer all four research questions, we needed to compare the test set quality obtained for the study object. For this, we used the rigorous, statistical testing approach proposed by Arcuri and Briand [10] that is widely used in the literature (e.g., [187]).

To indicate the significance in the differences of test set quality indicated by BC and strong MS for a pair of study objects, we used the non-parametric Mann-Whitney U test with a p-value threshold of 0.05 for hypothesis testing. In order to further strengthen the significance of the results, we complemented hypothesis testing with the Vargha-Delaney $\hat{A}_{12}$ estimate.

Strong Mutation Score From the $\hat{A}_{12}$ estimates obtained for strong MS and the pairs of study objects under comparison, we found that $TestGen_{2n}$ performs significantly better than —

- *RQ1a)*: $MonoGen_{2n}$ in 33 out of 120 classes (i.e., 27.5% of the classes) with an average improvement of 16% in mutation score (median 13%),
- *RQ1b)*: $MonoGen_{2}$ in 28 out of 120 classes (i.e., 23.3% of the classes).

For RQ1a), $MonoGen_{2n}$, on the contrary, is only significantly better than $TestGen_{2n}$ in 8 out of 120 classes (i.e., 6.6%). However, $TestGen_{2n}$ achieved an

average improvement of 16% in mutation score (median 13%), whereas $MonoGen_{2n}$ achieved a lower average improvement of 8% in mutation score (median 7%).

Branch Coverage In contrast to the performance results achieved for strong MS, from the $\hat{A}_{12}$ estimates obtained for BC and the pairs of study objects under comparison in RQ2a), we observed that $TestGen_{2n}$ only performs significantly better than $MonoGen_{2n}$ in 6 out of 120 classes (i.e., 5% of the classes), whereas $MonoGen_{2n}$ outperforms $TestGen_{2n}$ in 16 out of 120 classes (i.e., 12.4% of the classes). Similarly, for RQ2b), $TestGen_{2n}$ only performs significantly better than $MonoGen_2$ in 7 out of 120 classes (i.e., 5.8%).

A further analysis of the discrepancy in the results achieved for strong MS and BC for the study subjects under comparison revealed that BC levels were already high for a majority of the randomly sampled classes, meaning that $TestGen_{2n}$ does not have much potential to further increase them. In other words, $MonoGen_2$ already achieved high levels of branch coverage (i.e., half of the classes achieved BC equal or greater than 82%).

Regarding the magnitude of improvements for BC, we spotted lower average improvements than for strong MS which is in line with the already high branch coverage levels. $MonoGen_{2n}$, for instance, achieved an average improvement of 10% for the 16 classes it significantly improved (median 6%).

Summary

In summary, using LASSO we were able to define a novel AUTG approach, and to demonstrate that its prototype realisation, LASSO TESTGEN, offers practical benefits for end-users. LASSO TESTGEN outperforms EVOSUITE using the same computing resources and time budget for strong mutation score. Since strong mutation score is seen as the "gold standard" [6], improvements to mutations core are arguably more important than improvements to branch coverage alone.

The rich set of obtainable properties in the observatorium enabled us to explore and explain our findings for each research question in greater detail. The main finding of significant levels of diversity demonstrates that LASSO can actually help to explore and to exploit desired properties of repositories at the large scale.

15.4 Efficient Study Designs

The real-world study of LASSO TESTGEN demonstrates the strengths of the observatorium in two powerful ways. Firstly, in the previous section, the study results demonstrated that LASSO is a viable platform for building analysis services (i.e.,

LASSO TestGen) that solve (1) practical engineering problems, (2) can be more efficient than existing state-of-the-art approaches, and (3) leverage "big code" techniques in terms of the diversity of systems prevailing in large software repositories.

Secondly, we used the LASSO platform to realise "experimentation-as-a-service" at the same time in order to perform a study design of non-trivial complexity at a large scale. While the sample tool study walk-through in Section 15.2.1 demonstrates the core capabilities and benefits of the LASSO platform to develop efficient study designs, the real-world study of LASSO demonstrates the huge potential of the platform for real experiments.

Of course, studies can be performed today without the use of LASSO, but the required corpora and supporting harnesses etc. usually have to be prepared manually on a case-by-case basis. The main contribution of LASSO is to allow researchers to define executable study designs by writing a suitable study script in LSL. In the following section we discuss the main differences to state-of-the-art (manual) experimentation approaches.

15.4.1 Scripting

Our study highlights the fact that LASSO offers a unified approach providing a high degree of automation and integration of state-of-the-art tools, hence making it an effective alternative to "manual" study design approaches. Even though they involved a complex study setup, the two study pipeline scripts in Listing 28 and 29 only contain slightly over 300 lines of code (excluding the selection criteria, mainly consisting of index-based filters, see Section 8.2).

In manual study approaches, on the contrary, experimenters often need to rely on general-purpose scripting and tools (e.g., [83]) in order to realise their envisioned study design, possibly resulting in a huge number of "scattered" scripts, each of which handles a certain aspect of the study pipeline. Arguably, such "freestyle" scripts have serious drawbacks on the effectiveness of studies with respect to the overall study time budget available, including —

- limited interoperability between scripts and tools,
- limited robustness (i.e., unpredictability, especially in the presence of many unknown software systems),
- limited scalability.

Without a cohesive study pipeline design, experimenters have to execute each task in the study independently, making all the necessary steps disconnected, and thus the interoperability between tools (i.e., study objects and measurements) limited. This

has the consequence that all the required measures need to be manually collected and merged in order to analyse and interpret them. This is not only costly, but also error-prone (i.e., increases threats to validity). LSL scripts, on the other hand, provide a unified modelling approach to realise automated study pipeline designs. Once a pipeline has been established, a large number of study subjects can be processed automatically.

Since general-purpose scripting does not focus on experimentation, scripts written in languages other than LSL have limited robustness against the many unknowns and technical problems faced in large-scale studies, especially in the area of mining software repositories. For example, general-purpose scripts such as "standalone" Python scripts as used in [82, 81] can fail at any point in time if assumptions are not met. To ensure robustness, however, experimenters need to write code to handle any possible failures that might occur. Creating robust study scripts at the scale of large repositories is therefore a challenging endeavour. LASSO implements a variety of failure tolerance mechanisms. Since it realises the IoC pattern, for example, failed systems are automatically reported by LASSO and dropped from the pipeline. Of course, such failure handling mechanisms can be written in general-purpose scripting languages as well, but it costs time and effort to do it right. LASSO, on the other hand, already offers the required mechanisms "out-of-the-box", and they can be tailored to the needs of individual experiments.

Achieving robustness at the scale of computing grids is even more challenging. Since failures can occur on several computing machines, they have to be monitored and detected in a central way. Although researchers can use custom scripts to distribute their "work" over a set of distributed machines, they often have to establish manual ways to monitor their progress. The LASSO platform, on the other hand is distributed, and hence inherently scalable "by design", since it is built on a robust, distributed middleware. Experimenters can directly leverage the scalability of the LASSO platform and do not have to take scalability concerns into account, since most of them are handled explicitly. The platform scales both vertically and horizontally allowing more computing power to be achieved transparently by adding more machines.

15.4.2 Sampling

Automatically curating a corpus of executable systems with high precision is facilitated by LASSO CURATE. LASSO not only provides a unified modelling approach to integrate arbitrary repositories, it also attempts to ensure the executability of the systems they contain. Since obtaining executable systems is challenging for experimenters, they often curate their own corpus of executable systems manually

(e.g., SF110 [83]), resulting in labour-intensive tasks that may consume the majority of the time budget available for a study. The approach presented in this work not only makes the integration of new data sources more efficient, it also supports the integration of manually curated corpora to enable studies for replication purposes.

Selecting systems from a large search space requires "querying" support as well. As can be seen by our study pipeline, LASSO allows fine-grained selection criteria to be specified for sampling purposes using IDCS. In addition to selection criteria, the platform offers functions to realise certain sampling strategies that are required for studies in software engineering to achieve generalisability [20]. In our study design, we employed probabilistic sampling based on the selection of CUTs at random from a large search space (i.e., Maven Central). Custom solutions that are not based on sophisticated code search capabilities typically need to invest a lot of time to create the same functions for manually curated data sets.

15.4.3 Transparency and Controllability

The two study pipeline scripts in Listing 28 and 29 demonstrate that all the general components of the study design follow a systematic structure. The most important variables that are controlled in the study (i.e., operations) are contained in the preamble of the script and can be easily modified. In custom scripting solutions, controlled parameters may be scattered around a set of scripts, hence leading to decreased transparency and the increased likelihood of errors.

Ensuring both controllable and reusable execution environments and configurations is complex. Users of LASSO have fine-grained control over the execution environments used to make observations about systems (i.e., Java classes). In the LSL study pipelines they are explicitly defined, and they are reused among multiple tools (i.e., study objects and measurement facilities) in order to obtain valid data for comparison. Moreover, the same execution environments are deployed automatically in a distributed set of machines. If, for any reason, the execution environment needs to be adjusted, only the profile block in LSL needs to be modified, showing again the efficiency of LSL scripts.

As demonstrated by our LSL study pipeline, existing guidelines for rigorous experimentation in software engineering can be directly achieved using LASSO's expressive DSL to lower the threats to validity. In this case, the number of repetitions is implemented using a simple for-loop (i.e., two more lines of code).

15.4.4 Feedback-Driven - Iterative and Incremental Study Design

LASSO and LSL facilitate feedback-driven pipeline designs that enable —

- pilot studies (i.e., feasibility),
- iterative and incremental evolution of study designs,
- batching strategies.

Since potential study designs can be sketched in LSL, LASSO can provide rapid feedback on their feasibility. We sketched earlier versions of the study pipeline in order to check whether LASSO TESTGEN and the study design ideas are feasible. Moreover, we used prototype pipelines to identify sensible defaults for independent variables of the study (e.g., sampling criteria, execution environment etc.).

Since LSL scripts can be developed using agile practices (i.e., iterative and/or incremental development), they can be used to enable fast-feedback cycles in order to verify the overall pipeline and to ensure the creation of valid data. Further, this allows the study pipeline to be gradually extended (e.g., by adding more measurement actions), while checking intermediate results.

Whereas custom solutions may be designed to execute a study pipeline once, LASSO and LSL allow custom batching strategies to be defined. This allows researchers to obtain complete results on a subset of study subjects in order to guide their experimental process and to enable possible decision-making based on the analysis and interpretation of partial results. To this end, users of LSL can reuse existing study pipelines and limit them to a certain number of study subjects. More importantly, batching can be used to judge when to "end" the operation of a study in case a limited time budget is available.

15.4.5 Replication and Extensibility

The LSL scripts used in the realisation and assessment of LASSO TESTGEN provide three new opportunities. Firstly, they can be used to replicate our results (a) "as-is", (b) with different assumptions (e.g., different execution environments, different subjects etc.), or (c) they can be used as templates to further modify either the approach or the study design. Potential improvements to LASSO TESTGEN, therefore, can then be easily assessed again, by simply reusing the appropriate parts of the existing study design and pipeline encoded in the scripts.

Part VI

Epilogue

16 Related Work

This chapter discusses research work related to the software observatorium presented in this thesis. Since LASSO can be regarded as a mining platform, our work is generally relatable to the field of mining software repositories (MSR) [109]. Such mining approaches attempt to solve practical engineering problems by exploiting the wealth of domain information contained in (large) software repositories to derive (i.e., mine) new knowledge [103].

In the following, we first discuss related approaches for determining the behaviour of software systems. Then we elaborate on related work and research related to the constituent parts of the observatorium that were introduced in Chapter 4 to 11. Thereafter, we discuss related analysis platforms, followed by work related to the presented applications of LASSO (i.e., analysis services using it) and platforms that support software experimentation.

16.1 Program Analysis and Behaviour Determination

Historically, program analysis is typically associated with the static analysis community (cf. [183]) which focuses on topics such as compiler optimisation and program verification, whereas software testing, debugging and profiling is typically associated with the dynamic analysis community. This thesis however, takes a holistic view on program analysis and uses it as an umbrella term to include static analysis approaches, dynamic analysis approaches and hybrid combinations of both [76]. In the following, the terms “program” and “software system” are used interchangeably to discuss the notion and role of behaviour.

Program analysis, in its basic form, is concerned with automatically examining the behaviour of a program to understand properties like correctness or safety (e.g., to discover program vulnerabilities). Even though dynamic and static analysis approaches both focus on the possible “executions” of a program, in the former the program of interest is actually executed by the underlying computing platform whereas in the latter it is not. Static analysis approaches only examine the source code of a program (i.e., its syntactic structure) to judge its run-time behaviour by reasoning over all possible executions. Dynamic analysis, on the other hand, is defined as “the analysis of the properties of a running system” (cf. Ball [19]) by executing it with a subset of all possible stimuli.

Neither static analysis nor dynamic analysis can solve the core undecidability problems in computability theory. In 1936, Alan Turing proved that over a program and its input, it is undecidable whether it will ever finish or run forever [242]. Similarly, Rice's theorem states that all non-trivial semantic properties of programs like behaviour are undecidable [201]. In practice, the core problem underlying both forms of analyses is the large, and often infinite, number of possible executions of the program of interest.

Since static approaches require significant resources and time to analyse non-trivial programs with a large number of possible states, they rely on models to reduce the number of states by abstraction (e.g., data-flow analysis based on graph abstractions). The core challenge is to pick a suitable model that achieves a useful reduction in states without sacrificing too much accuracy (i.e., without abstracting away too much relevant information). The presence of dynamic language features whose semantics are only exhibited at execution time, like reflection in Java, make it particularly difficult for static analysis techniques to reason over all possible executions of a program [38, 160].

Static analysis techniques, therefore, strive to be conservative in order to preserve and satisfy the important property of "soundness" which is achieved if all reported properties of the static analysis are always true. Conservatism makes it possible to define weaker properties which are more likely to be true than stronger properties. As a result, static analysis approaches often produce (over-)approximations that sacrifice precision.

Since dynamic analysis, in contrast, executes the program on a computing platform, its actual (i.e., true) behaviour can be observed for specific stimuli, even in the presence of dynamic language features. This results in high precision observations, since no abstract interpretation is involved as in static analysis, but the downside is that only a subset of all possible executions can be analysed, even for small programs. Dynamic analysis approaches are therefore inherently incomplete [55], and may lead to a subset of observations that may not be sufficiently generalisable to characterise the actual behaviour of a program. They can therefore be characterised as following a goal-oriented strategy. Typically, representative sets of stimuli (i.e., tests) are selected that exercise certain portions of the behaviour of the program under analysis to gain insights into its run-time properties. Observations about the behaviour of a program can usually be obtained as quickly as the program can be executed, but is often costly since it is necessary to —

- "instrument" the program, which may negatively affect its run-time performance,
- record potential large execution traces in a scalable manner,

- find efficient ways to store traces in order to process them "offline".

Other research challenges relate to the kind of program entities analysed in dynamic analysis approaches [76]. Sometimes it is sufficient to only cover entities like methods, statements or branches of the program "once", but in other cases it is necessary to cover entities more frequently (i.e., n times). This has a bearing on how costly the analysis approach is with respect to time and memory, and how accurate its results are. The cost and accuracy of dynamic analysis is also partially related to the instrumentation techniques applied and the kind of information collected. Also, the "observer effect" [55] which occurs, for instance, in parallelised programs that use the multi-threading capabilities of the hardware, may result in different observations from the same repeated executions of the program. This makes the attainment of consistent analysis results challenging.

Both kinds of analysis have pros and cons, therefore. Static analysis approaches typically have the advantage of soundness, but at the expense of over-approximation (i.e., precision) due to the need for abstraction whereas dynamic analyses have the advantage of precision, but at the expense of under-approximation (i.e., soundness). Dynamic and static analysis approaches therefore have the potential to complement each other and create synergies (i.e., by combining sound static analysis with precise dynamic analysis) [76]. For instance, by using dynamic analysis results as input for static analysis approaches, or vice versa, more powerful analysis results can be obtained in which the strengths of one compensate for the weaknesses of the other.

16.1.1 Equivalence Checking

A subfield of program analysis concerned with establishing whether two functions or programs are functionally equivalent is "equivalence checking". According to Churchill et al., the problem of equivalence checking is to *formally prove* whether two software systems are "semantically" equivalent [51, 210]. This is considered a long-standing problem in compiler correctness (e.g., optimisations), superoptimisation (code optimisation), program synthesis as well as code refactoring correctness.

Since this field of research focuses on approaches for proving functional equivalence at the level of granularity needed to address the aforementioned problems, they do not (yet) scale to the needs of the observatorium (i.e., because they typically use costly trace-based analyses). The major objective in equivalence checking is to compare the behaviour of two "homogenous" programs/functions where one has been derived from the other (e.g., through refactoring or optimisation). In the context of our work, at the scale of repositories, the observatorium usually needs to compare separately developed, "heterogeneous" systems that possibly require "adaptation" (Section 9.3).

16.1.2 Practical Implications

As discussed above, dynamic analysis is preferable in cases where precise and efficient analyses are required, since it does not rely on expensive abstractions. However, this advantage comes at the cost of limited "soundness". Although the results are more detailed, they are specific to the executed tests and are thus not generalisable to all possible executions of the program. In contrast, static analysis can prove certain properties like correctness for all possible executions (e.g., theorem-proving), whereas dynamic analysis can only provide observations about the presence of expected behaviours with respect to a given set of stimuli in a certain execution environment. Dynamic analysis cannot show the absence of unexpected behaviours (in accordance with Dijkstra's famous quote: "Program testing can be used to show the presence of bugs, but never to show their absence!" [69]).

In the context of the observatorium developed in this thesis, dynamic analysis (i.e., software testing) is a practical way to observe and locate interesting behaviour in a precise and efficient way, even at a large scale. The core challenge here is to carefully select test sequences (i.e., stimuli) to increase confidence in the presence of desired behaviours (see behaviour sampling in Chapter 9). It is stressed, however, that although dynamic analysis approaches are the mainstay of the observatorium, static analyses techniques are also used to complement them. Results from static analysis, for example, can be used to improve the efficiency of dynamic analysis by "pre-selecting" software systems that likely exhibit the desired functionality (i.e., by using IDCS based on static code analysis and NLP techniques).

16.2 Constituent Technologies

In this section, we discuss work and research related to the main component technologies developed in this thesis to realise the observatorium.

Sequence Sheet Notation

The sequence sheet notation (SSN) is inspired by test sheets [14, 13] which, in turn, are inspired by the idea of FIT tables and FIT test definitions [178]. One of the driving forces behind test sheets was to support the creation of test specifications which are (a) executable, and (b) bridge the conceptual gap for the engineers who create them. Like sequence sheets, test sheets use a spreadsheet-like notation to describe a sequence of operations. However, sequence sheets remain more faithful to the features of object-oriented languages. Whereas test sheets define a sequence

of service operations, for example, sequence sheets define method invocations of functional abstractions and (concrete) classes.

Test sheets support the definition of tests that used conditional expressions (e.g., to model pre- and post-conditions of service contracts). When a test sheet is executed, a result test sheet is created that shows the results in a way that resembles the "traffic light" notation known from unit testing practices (i.e., red for failed and green for successful assertions). In the observatorium, on the other hand, sequence sheets are executed using the arena and are individually analysed "offline" using the observatorium's data analytics layer (Chapter 6.7).

The most important difference to test sheets, however, is that the executed sheets (i.e., actuation sheets) store stimulus/response pairs that can be analysed later in data-driven ways. Since sequence sheets are used to populate the cells of SRMs, and define simple sequences of method invocations, observed stimulus/response pairs are universally accessible in a well-defined navigation model. Test sheets do not offer this capability.

Stimulus Response Matrices

At the present time there are no approaches for assembling and storing large collections of system stimuli and responses similar to the SRM approach. The formal model underpinning SRMs is loosely inspired by Barr et al.'s abstract stimulus response model [24] used to describe oracle approaches. However, this approach does not offer a practical language or data structure for representing stimuli and responses at a large scale.

In practice, SRMs are loosely related to collections of units that are tested by a set of accompanying classes as in unit testing frameworks. The reporting features of unit testing frameworks, in this case, serve as the primary approach for storing structural representations of the test execution results. However, since these reports do not record observational records[1] (i.e., analysis attributes such as stimulus/response pairs), they cannot be used for data-driven analyses of system behaviour. This is because they "mix" method invocations with stimuli and expected responses in an "ad hoc" way within test cases.

Creating, Storing and Accessing SRMs

The observatorium provides advanced services to analyse SRMs in a data-driven way based on the OLAP paradigm. A large number of analytical platforms can be regarded as instances of the OLAP approach, but not always in the traditional

[1] Apart from potential assertion failures that may come with some human-readable description.

sense of providing a full-blown analytics platform that explicitly supports data warehouses, data lakes and cubes. We use the terminology established by OLAP to motivate our approach to the data-driven analyses of SRMs. Technically, however, we support analytical operations of OLAP cubes based on state-of-the-art statistical (data mining) platforms such as R, PANDAS and APACHE SPARK which all use *data frames* to represent and manipulate data efficiently.

Based on this representation, OLAP cube operations can be realised in a variety of ways, even though the OLAP terminology is not explicitly adopted for these kinds of tools. For example, statistical tools like R provide a rich set of libraries offering a variety of operations to realise cube functionality. In particular, the TIDYVERSE ecosystem [240] offers powerful operations to group (roll-up), filter (drill-up/down, slice and dice) and widen/lengthen tables (pivoting).

In the same vein, in contrast to mature fields like business intelligence, the envisaged observatorium does not always require a fully-blown data warehouse using popular modelling techniques like snowflake schemata. Instead, the aforementioned tools support efficient complex, "ad hoc" queries even on single machines with moderate computing power.

Even though the different tools use the same data frame structure, however, they use different approaches for realising them. To support the efficient manipulation of data frames at a large scale, APACHE SPARK offers a data frame implementation that scales in a cluster of nodes (based on resilient distributed data sets (RDD) [235]. R and PANDAS are designed to run on single machines, but they scale (horizontally) as well with respect to local parallelism (i.e., multi-threading). Vertical scaling is also possible to a certain extent by using third-party extensions.

The languages used to manipulate data frames also varies from tool to tool. While APACHE SPARK primarily uses Scala/Java to manipulate data frames, R uses the R programming language and PANDAS the Python programming language. Most interestingly, all of these basically provide "bindings" (or bridges) to support other languages as well (e.g., data frames from APACHE SPARK can be used within the R language). Overall, the aforementioned tools are well-integrated into an ecosystem of machine learning pipelines so that results from data frame manipulation can be used to feed machine learning pipelines that employ state-of-the-art techniques like classification and prediction etc.

System Boundary Model

The measurement of scopes explained in Section 5.3.2 has similarities with program slicing techniques [250] and slice-based metrics [175]. Program slicing has been proposed to support engineering activities such as debugging, for example. The

core idea behind program slicing is to compute a slice of software components of a single system that are affected by a slicing criterion based on a representation of the system's code elements as a system dependence graph (SDG) [116] that combines several procedure dependence graphs (PDG). Slice-based metrics are then defined based on the SDGs.

Using this terminology, our system scoping approach involves two consecutive "slicing" steps, each of which produces a set of software components. First, the determination of the boundary of the software components of a system is related to the idea of interface slicing [32]. Based on the entry methods of a system (determined via the interface of the functional abstraction), static call graph analysis is used to return all (transitive) methods involved in the behaviour of interest from which we then approximate a set of software components. Second, in an approach that is loosely similar to dynamic slicing [254], custom scoping criteria are then applied to reduce the set of software components to those which are of interest to the user.

The main difference to program slicing is that our measurement model is not limited to the representation of SDGs and the criteria/metrics applicable to them. Instead, users are able to select their own metrics and to specify their individual scoping criteria that decide which system components are of interest.

Executable Corpora

A variety of general-purpose, executable corpora have been proposed in software engineering research. For example, XCorpus [68] offers a set of "76 executable, real-world Java programs, including a subset of 70 programs from the Qualitas Corpus" (cf. [229]) in order to support experimentation in static and dynamic program analyses. Similarly, 50-K [172] offers a data set of $50,000$ compilable Java projects including their build scripts to foster experimentation. NJR [185], on the other hand, "envisions" a set of $100,000$ executable Java programs which can be obtained via cloud services to support the development of new (academic) tools and techniques. Even though these executable corpora share the same vision underpinning our observatorium concept, they are limited with respect to the provided tooling, the underlying model and the scale of the repository (i.e., they use ad hoc curation techniques). Our approach aims to provide an integrated approach that allows the *automatic* curation of behaviour-aware, "live data sets" [68] from any data sources of interest, potentially at an ultra-large scale (see Chapter 8 and 9). Here LSL is used as a powerful dynamic query language to express non-trivial curation criteria.

SF110 and DEFECTS4J are examples of (manually curated) executable corpora developed for a given purpose. Their creation required huge effort as described

by Problem *P1*. SF110 contains a manually curated set of 110 Open Source Java projects to facilitate studies of the effectiveness of automated unit test generation algorithms and tools [83], DEFECTS4J [133] contains classes from 17 Open Source Java projects that contain 835 bugs. Its aim is to provide "a database of existing faults to enable controlled testing studies for Java programs".

Finally, an example of a non-executable corpus containing "incomplete" code that is unlikely to be executable is BIGCLONEBENCH [226]. This corpus aims to support the evaluation of clone detection techniques and tools and contains eight million validated clones sourced from 25,000 Open Source projects. Since the corpus focuses on code clone detection only, it primarily aims to facilitate static program analysis techniques, so the executability of the code clones is unimportant. There is, consequently, little if any information about the executability of its contents.

As demonstrated in Chapter 12, LASSO has the capability to integrate existing, manually curated software engineering corpora thanks to its unified repository modelling approach (Section 12.4).

Code Search Engines

As part of the rise of the Open Source movement [159], research on code search engines started to kick off in the 2000s when the number of code-related artefacts retrievable over the Internet reached a critical mass [118]. The interest in code search engines has expanded rapidly as can be seen by 80% of the $\sim$ 100 papers published on code search engines since 2008 [101]. The main objective of code search engines has been to facilitate software reuse [149, 176]

To date, coping with the vast, and rapidly increasing, number of code artefacts stored in Internet repositories (e.g., 85 million new projects were created on GitHub in 2022 alone), is one of the core challenges addressed by most of the current research on code search engines. Leveraging this huge number of code artefacts at a large scale is therefore a natural "big data" problem.

In the remainder of this subsection, we discuss work related to the text-based selection and test-driven selection capabilities offered by the observatorium. A more comprehensive and recent overview on the techniques applied for "finding code" is provided by Grazia and Pradel [101].

Text-Based Selection

The text-based retrieval of artefacts is a large research area with applications in a wide range of domains. An overview of information retrieval techniques for general-purpose, text-based search engines can be found in [168], while an overview of query expansion techniques that attempt to further improve recall can be found

in [45]. More specifically, in the area of software engineering, Sim et al. give an overview of text-based retrieval techniques in code search engines [212], and Robillard et al. [204] provide an overview of these for recommender systems.

IDCS was inspired by the idea of signature matching from Zaremski et al. [256], but was most comprehensibly implemented by Hummel [117] as part of the MERCOBASE code search engine that aimed to facilitate software component reuse. In our approach, LQL basically extends and enriches Hummel's MQL query language for IDCS with useful extensions to further improve recall (Section 8.4.1). Lemos et al. also provide an approach for IDCS as part of SOURCERER which, like LASSO, employs an automated query expansion technique based on WordNet [158]. All approaches share the fact that they represent interfaces and estimate relevance using bag-of-words models [168]. In contrast, Nie et al. [182] developed an automated query expansion approach based on crowd knowledge which sources vocabulary from *StackOverflow* based on the idea of mining domain-specific vocabulary from questions and answers. Our approach uses WordNet together with an English dictionary that is not specific to any problem domain. We plan to integrate domain-specific vocabularies in LASSO in future iterations of the platform. Moreover, we also plan to leverage the capabilities of LASSO to create new vocabularies.

Test-Driven Selection

One of the main weaknesses of most contemporary code search engines is their reliance on text-based approaches for selecting software systems such as IDCS (Section 8.3), which means they are only able to estimate the run-time behaviour of software systems from the identifiers used to name classes, method, parameters and their types. Although these estimation techniques have become quite sophisticated by applying NLP techniques such as word stemming, query expansion (e.g., [182]), topic modelling and AI techniques like neural networks (e.g., [102, 49]), they can never fully overcome the idiosyncratic choices of software engineers when selecting identifiers (cf. vocabulary mismatch problem [87]).

The only practical way of overcoming the influence of identifier choices in estimating the functional equivalence of software systems is therefore to compare their responses to a sample set of stimuli from their input space. This approach, first proposed under the name of "behaviour sampling" in the early 90s, has been shown to be effective provided the set of stimuli is of sufficient size and quality [191, 141]. In an effort to support true, behaviour-aware searches alongside textual searches, several code search engines and recommendation systems incorporate some form of behaviour sampling capability under the name of test-driven (code) search. However, the majority of these engines and systems are no longer maintained.

To avoid significant reductions in recall, a key component of a test-driven (code) search engine is effective software adaptation to consider systems whose interfaces may not directly match that of the sought-after functional abstraction. Early attempts at systematic software adaptation date back to the era of component-based software engineering where "software components" needed to be adapted to integrate them into new architectures [43, 41, 34]. Adaptation is also a problem tackled in the field of software evolution (e.g, API usage and evolution [181, 57]).

In the context of test-driven search, the S6 engine "reassembles" software components in such a way that they conform to a set of tests specified in terms of input and output values [200]. Wang et al. [249] propose the use of linear optimisation for interface signature matching in the context of Reiss' S6 engine [200]. CODEGENIE/SOURCERER, on the other hand, use JUNIT test classes to return slices of code elements [154, 155, 17]. Similarly, MEROBASE/CODECONJURER uses JUNIT tests as test sequences, but returns the entry classes that were matched via IDCS [117, 120]. The platform uses a brute-force approach in an attempt to try all possible adaptations using a small set of adaptation operators (parameter switching and type relaxations).

The adaptation framework proposed in this thesis is included as a first-class citizen of the observatorium as part of its test-driven selection service and can be configured on a case-by-case basis. It is more powerful than a brute-force approach, since it is based on an extensible set of adaptation operators that increase the likelihood of potential interface matches. To this end, it uses an opportunistic prioritisation ranking scheme to first execute the candidates that are most likely to be relevant.

In contrast to the aforementioned test-driven search engines which rely on dynamic analysis, SATSY [221] uses a different approach based on "static execution". It uses static program analysis in terms of symbolic execution and constraint solving [18] under the hood to establish whether the behaviour of one or more software systems match. Although this approach is promising, the current limitations of constraint solving mean it is currently only applicable to tiny code snippets. Since no traditional execution on a computing platform is performed, additional measurable engineering goals such as run-time characteristics in certain execution environments (cf. run-time profiles and scope-aware measurements in LSL) are not supported by this technique (e.g., performance-related properties such as execution time).

The aforementioned approaches to behaviour sampling are unsuitable for an observatorium working at the scale of big code, since well-defined software systems and well-defined test sequence representations are required to support test-driven selection at a large scale. There is no clear definition of what constitutes a system in terms of its code elements, and the test sequences used are sometimes limited to single input/output value mappings that can only test for utility functionality [155]

(since the actuations are hidden from the user). There is also no clear separation between stimuli and responses. Approaches that adopt JUnit test classes as test specifications suffer from the same limitations as mentioned in Problem *P4* (Section 1.2).

Finally, the FaCoY search engine was developed to support code-driven search [144] using sophisticated static analysis, but compared to observation-aware, code-driven selection, it is still limited by the intractability of establishing functional equivalence.

Advanced Selection Criteria

LASSO provides a holistic measurement approach for formulating selection criteria based on arbitrary system properties. Compared to existing mining platforms such as Boa [75] or QualBoa [67], LASSO is not limited to properties statically inferred by AST analyses, but also supports properties dynamically measured from individual systems as well as collections of systems (e.g., implementational distinctness and diversity). The measurement of properties, as well as the "enforcement" of properties, is well-integrated into the observatorium by means of its data structures, analysis architecture and system boundary model.

Reiss' S6 [200] uses single, size-based metrics (i.e., complexity) to rank system matches in test-driven searches. QualBoa [67] based on Boa, and SOCORA [142, 143] also provide software component ranking approaches driven by software metrics indicating non-functional properties of systems (i.e., software quality). Neither approach, however, is true-behaviour-aware, since they measure metrics on static scopes of systems (in contrast to the behaviour-aware, scope-based measurements supported by LASSO).

Inspecting Software System Quality

One of the capabilities of the LASSO platform is to allow users to inspect and monitor the quality of many software systems efficiently at a large scale based on the definition and automation of custom measurements (see Chapter 5). Work released under the term *software tomography* also attempts to realise efficient inspections and measurements of software quality. Firstly, Bowring et al. present the Gamma tool that aims to facilitate efficient, scalable remote monitoring of software deployed in the field in order to react to software failures as quick as possible [40]. The authors were inspired by the core idea behind tomography (as known from radiology) in order to scale the costly task of software monitoring and instrumentation over many instances by splitting up the task into suitable subtasks, each of which introduces

only a small, "acceptable" (instrumental) execution overhead to each deployed software instance. Since the task of instrumentation is split over multiple instances, software tomography needs to deal with the efficient collection and aggregation of partial monitoring data in order to gather all monitoring information.

In a sense, the observatorium deals with similar scalability challenges in order to allow for custom analysis steps, but also needs to scale to many instances of many software systems. Similar to the idea of tomography, analyses are split up into manageable subtasks using the concept of LASSO actions that are used as part of LASSO's analysis pipelines. The analysis architecture presented in Section 6.5 offers a scalable distributed architecture in order to collect and aggregate analysis results gathered from worker machines for later interpretation.

Another interpretation of the notion of software tomography relates to the idea of making hidden facts (i.e., quality aspects) of software systems visible to their stakeholders. In this context, software tomography is applied to extract data, visualise and interpret it [21, 22] in order to facilitate reengineering tasks that aim to improve the quality of software systems [213]. The commercial tools SOTOGRAPH and its successor SONARGRAPH [121] are rich static code analysers that allow stakeholders (e.g., software architects and developers) to "monitor" software systems with respect to their quality aspects, especially by defining custom rules and measuring metrics in a continuous manner.

Like LASSO, they offer scripting capabilities (i.e., a DSL) to formulate custom analysis rules and metrics, but they are limited to static analyses of software systems. Primarily, these tools focus on the analysis of mostly single systems that stakeholders know. The arena component and the concept of SRMs of the observatorium scales to the needs of mass analysis of software systems that are likely unknown to users. Moreover, it is not the aim to replace the aforementioned tools, but to integrate them into the LASSO platform to further enrich the analysis capabilities provided by the observatorium to its users.

Similar to SOTOGRAPH and SONARGRAPH, SONARQUBE [216] enables the monitoring of code quality based on a set of static code analysers, rules and (custom) "compound" metrics in a continuous manner. Again, the LASSO platform does not aim to replace such platforms, but fosters their integration to offer users more (technical) possibilities for realising certain (static) code analyses in LASSO.

Similarity, Redundancy and Diversity

A field that is closely related to the comparison of software systems and that also involves the notion of "similarity" is software redundancy. The terminology used in redundancy-related fields of research is often inconsistent and misleading, especially

when the notion of code clone detection [207] is brought into the picture. The notions of "equivalence" and "similarity" are particularly affected, since these two fields (software redundancy exploitation and code clone detection) differ in the kind of information they inspect to establish whether two systems are deemed to be redundant (e.g., either by comparing software components or behaviours).

Code clone terminology is not even used consistently within the code clone detection community, but there is general recognition of four basic clone types based on the notion of some (vaguely-)defined similarity notions [207, 36, 148, 206, 198, 2] —

- *type-1*: textual similarity,
- *type-2*: lexical similarity,
- *type-3*: syntactic similarity, and,
- *type-4*: functional similarity, but syntactic dissimilarity.

Type-1 to type-3 clones rely on textual similarity measures (on which most clone research has focused), whereas type-4 clones rely on some kind of functional similarity measure. The biggest drawback of this terminology is that although the types classify clones, they do not define what a clone is. Gold et al. state that "[t]his is usually defined to mean that two source code fragments are clones if they are similar with respect to some defined similarity measure" [96].

So neither code clones nor their similarity measure are clearly defined by this terminology. Several authors have therefore proposed further subtypes such as exact clones (type-1), renamed clones and parameterised clones (type-2), near-miss clones (type-3), and semantic clones (type-4) based on some degree of similarity.

The notion of type-4 clone comes close to our notion of heteromorphic redundancy (i.e., functionally equivalent, but implementationally distinct, see Section 14.1). However, it is vaguely defined and thus provides significant leeway for different interpretations. For example, Roy and Cordy define type-4 clones as: "Two or more code fragments that perform the same computation but implemented through syntactic variants" [207]. However, there seems to be no common agreement in the community as to what "same computation" is and what "syntactic variants" are, so it is not clear what differentiates a type-3 from a type-4 clone [226, 225, 253].

Since similarity measures are essentially undefined in the classification of clone types, functional similarity (or semantic similarity) is also open to multiple interpretations [131]. Roy and Cordy [207] define functional similarity for two code fragments as functionally identical or similar, Jiang and Su [128] as functionally equivalent, and Saini et al. state that in the case of type-4 clones "the goal of clone

detection is similarity, and not exact equivalence (including for semantics)" [208]. In contrast to functional equivalence, Gabel et al. consider two code fragments with isomorphic program dependence graphs to be clones [88] which may not be functionally equivalent with respect to their actually observed functional behaviour [131].

Since the 1980s, when Avizienis et al. first advocated N-version programming to improve the reliability of embedded systems [16], software redundancy has been exploited in many areas of software engineering, from software reuse and code recommendation [144] to test automation [46]. However, obtaining redundant software of the right kind and scale can be difficult, depending on the properties and level of granularity required [89, 128]. The classic notion of redundancy requires implementational distinctness, whereas code clone detection favours implementational similarity [148]. Since in our work simple redundancy requires functional equivalence, heteromorphically redundant systems that are also implementationally distinct most closely match the weakly type-3 and type-4 clones subtype (cf. BigCloneBench [224, 225]).

16.3 Big Code - Platforms, DSLs and Data Sets

Undoubtedly, the LASSO platform shares many traits with code search engines and their selection criteria. Requiring the results of a code search to have the desired functionality is clearly one of the most important search criteria, however, there are many other factors that can influence the relevance of the search results. These can range from simple size-based code metrics such as cyclomatic complexity to more complex measures such as the degree of implementational distinctness. Since the determination of these measures can be quite complex, code search engines ideally need to support the application of sophisticated analysis algorithms at a large scale (i.e., the big data scale). Moreover, to allow users to choose new relevance criteria, they need to be able to specify new kinds of analysis algorithms in an abstract way without having to code up complex new algorithms. The current generation of code search engines, however, support few higher-order criteria and certainly do not allow new measures to be defined by users in an abstract manner.

Mining Platforms and DSLs

The provision of precisely this kind of capability is the focus of the relatively new field of large-scale software analysis platforms such as BOA [75] and SOURCERER [17]. The goal of these platforms is to gather a huge (sometimes called "ultra-large") repository of software components and make them analysable in an abstract way

through a dedicated, high-level, domain-specific language. BOA was not specifically designed to support interface-based and test-driven code search, but it can support arbitrary queries over the abstract syntax of code units. Moreover, as demonstrated by QUALBOA [67], additional analysis capabilities can be used to define reuse-oriented relevance ranking algorithms [67]. On the other hand, SOURCERERCC [209], an extension to the SOURCERER platform, supports syntactic code clone detection at a very large scale. The weakness of these large-scale analysis engines, however, is that their abstract, domain-specific languages do not accommodate dynamic (i.e., execution-based) algorithms and metrics. Moreover, even when they do include dynamic properties such as GREENMINER [113], they do not pursue a holistic approach, and thus only offer certain services of a software observatorium.

LASSO provides a unified platform to deliver analysis services that accommodate a variety of rich, multi-criteria system comparisons based on LSL as a dynamic query language to encode individual, behaviour-aware selection criteria. LSL is inspired by data-flow programming languages [129]. In the context of our work, practical examples of workflow DSLs include the "Gradle Build Language" of Gradle [99] that is built around the notion of a "task" (similar to LASSO actions), and Jenkins' Pipeline DSL [126] that is used to manage continuous integration pipelines. LSL even incorporates an extended version of MQL used by MEROBASE in terms of LQL, so it provides more powerful options than MEROBASE with respect to text-based selection.

LSL's unified stimulus/response model allows the development of new dynamic algorithms and metrics, on the one hand, and its pipeline of actions allows for the integration of existing algorithms and metrics on the other hand. Moreover, in order to increase the accuracy of goal-oriented measurements, scope-awareness facilitates the fine-tuning of software metrics to a restricted set of software components on a case-by-case basis.

Data Sets of Version Control Data

The exponential growth of Open Source software projects managed by public version control systems, mainly in GitHub (cf. [94]), resulted in multiple initiatives that aim to offer researchers convenient access to version control data in order to facilitate their mining activities and engineering studies. One of the main objectives is to link and query the development history mined from version control systems like Git (e.g., commits, tags and social data like developers etc.).

GHTORRENT [98], for instance, collects event-driven data from GitHub's web service and makes it available as a database dump to researchers. The "public Git archive" [170] offers a big code data set of "$182,014$ top-bookmarked Git repositories

from GitHub" including their development history. CODEDJ [166], on the other hand, offers a query interface over large-scale software repositories, similar to BOA's DSL. World of Code (WoC) [165] offers an expandable infrastructure to mine version control data and code from large repositories at scales that allow (research) questions of "global reach" to be formulated.

All these data set initiatives share the fact that they facilitate the access to huge amounts of version control data, and optionally code, by proposing efficient crawling and retrieval architectures. Compared to LASSO, there are two major differences, however. Firstly, in this thesis we developed an approach to query and curate data sets of executable systems, whereas the aforementioned approaches mainly aim to retrieve version control data and code, without further analysing it. They neither provide code search engine selection capabilities like IDCS, nor analysis capabilities of the kind through LSL. Secondly, LASSO's executable corpus is code-centric and does not focus on version control properties of software systems (so far). A natural addition to LASSO is to integrate those external data sets as "data sources" into the platform to allow the querying of version control data about classes of interest.

Finally, TRAVISTORRENT [35] is another data set approach that aims to also provide metadata about software projects, but in this case it is mined from the reports generated by the continuous integration system TravisCI.

16.4 Analysis Services

The ability to develop new analysis services on top of the observatorium is novel. Related platforms merely offer a "family" of related artefacts, rather than a fully integrated platform.

16.4.1 LASSO Search and LASSO Curate

Code search engines related to LASSO have already been discussed in terms of their search capabilities (either text-based, test-driven or both). Other tools that are related to the IntelliJ IDE plug-in offered by LASSO SEARCH are CODEGENIE [154] and CODECONJURER [120]. Both tools, which are companions of the search platforms SOURCERER and MEROBASE, respectively, provide reuse recommendations driven by test-driven code search services in the IDE. Our plug-in, however, offers many more search scenarios based on the rich selection criteria offered by the LASSO platform. In addition to test-driven selection, our tool integration enables code-driven searches as well as the quality-aware selection of candidate systems based on the (advanced) selection criteria provided by the platform. Moreover,

custom reuse criteria can be optionally specified using custom LSL scripts as a dynamic query language.

We already discussed (executable) corpora related to the curation capabilities offered by LASSO CURATE. At the time of writing, LSL as a dynamic query language to express rich curation requirements, is a novel technology, and we are not aware of any similar approaches.

16.4.2 LASSO TestGen and LASSO TestAmp

The software testing services introduced in Chapter 14 make a contribution to the field of AUTG and test amplification. An important research problem in these fields is the criteria used to indicate the quality of the tests that are returned. Whereas a recent overview of test amplification approaches can be found in [58], AUTG and test coverage criteria are discussed below.

Automated Unit Test Generation (AUTG)

AUTG tools have the ability to generate high-quality test sequences solely using a realisation of a functional abstraction of interest (i.e., code of the system under test). The most effective test generation approaches employ search-based algorithms (cf. SBST [174]) which use randomly generated "seeds" to search for optimal test inputs based on one or more "meta-heuristic" fitness functions. Fitness functions are defined based on the test selection criteria which is being optimised (e.g., branch coverage and mutation score, see Section 16.4.2).

At the time of writing, the most effective tool for creating Java unit tests, at least as a research prototype, is EVOSUITE which has frequently won the SBST tool challenge (e.g., [246]) and has regularly been shown to outperform other tools [83]. The most recent version of EVOSUITE, enhanced to use a novel many-objective search algorithm [187], achieves average coverage scores of 74.5% and 76.4% for line- and branch coverage on Java classes.

Another popular automated unit test generator is RANDOOP that is driven by feedback-directed random test generation, a variant of random testing [184]. The strategy behind this special random testing technique is to generate, shuffle and reassemble test sequences for a Java class in a clever manner. Test sequences are randomly evolved, but each iteration is based on the results (i.e., feedback) from the former iteration.

Both search-based and random-based AUTG have been demonstrated to detect faults in real-world Java classes. The authors of EVOSUITE have shown that search-based test unit generation is, in general, superior to random-based test unit generation [83].

AUTG tools, however, are limited. Firstly, they produce test sequences solely based on the structural coverage of the system under test and are "agnostic" to the system's specification (i.e., functional abstraction). Secondly, AUTG does not solve the oracle problem [24], so each generated set of test sequences needs to be manually checked by engineers based on their knowledge of the functional abstraction under test. The only way AUTG tools help in this case, is by recording the response of the systems when they are stimulated in order to enable regression testing (i.e., comparing the execution of the previous version of the system with a new version when changes are introduced). Finally, a fundamental research question which is still "open" is "does automated unit test generation really help software testers?" [86].

Test Set Quality Assessment

The aim of software (unit) testing is to discover defects in the system under test in order to improve its quality in terms of reliability. In order to increase the confidence that a set of test sequences (sometimes referred to as a test suite) really has a positive effect on the reliability of the system, several test coverage metrics (i.e., test coverage criteria) are used to measure a form of "coverage", often expressed in terms of a score (ratio) [6].

In practice, as well as in academia, perhaps the most widely used of these are —

- *Line Coverage* which is the percentage of lines exercised by a test set relative to the number of lines in a given class,
- *Branch Coverage* which is the percentage of the branches exercised by a test set relative to the total number of branches in a given class,
- *Strong Mutation Coverage* which is the percentage of the mutants "killed" by a test set relative to the total number of non-equivalent mutants (i.e., mutation adequacy score) for a given class.

Line- and branch coverage measure the number of code elements executed by a set of test sequences. Such code coverage measurements can be defined for virtually any type of code elements, including methods, exceptions, and statements etc.

Mutation testing is considered to be the "gold standard" in software testing that subsumes several other test coverage criteria such as code coverage [6]. The general idea of mutation testing is to seed faults into the software [8, 188]. The process creates mutants of the original system under test based on a set of mutation operators that change the syntactical elements of the system. To generate a mutant, typically one mutation operator is applied to one location of the system's code. Mutation operators are designed to resemble typical "simple faults" made while programming

(e.g., flipping an operator from "+" to "-"). Based on the *competent programmer hypothesis (CPH)* and the coupling effect, simple faults are thought to also effectively uncover more complex faults [65].

A common strategy to generate a set of mutants (i.e., to seed all possible faults) is to apply all operators to all possible locations. The main objective in mutation testing is that at least one test sequence is able to detect the seeded fault (i.e., cause the mutant to exhibit different behaviour to the original system), which is referred to as "killing" the corresponding mutant. Based on the ratio of the number of killed mutants to the total number of mutants to be killed, a mutation score is calculated. An existing research challenge is the detection of equivalent mutants, namely the case in which two variants of the system under test behave the same, even though their code elements are not identical (e.g., when flipping an operator does not change the underlying behaviour of the system in terms of its responses). The detection of equivalent mutants in practice, however, is non-deterministic because of Rice's theorem. Strictly speaking, the mutation scores computed by state-of-the-art mutation testing tools like PIT are imprecise, since the generation of equivalent mutants cannot be ruled out entirely. More future research is necessary to improve the detection of equivalent mutants.

The measurement of test quality metrics directly depends on the particular code elements in the system under test. The existing landscape of tools and techniques used, however, all have their own assumptions about the level of granularity of code that is considered and the "extent" of code elements that are included in the analysis. In Chapter 5, we propose that the boundaries of software systems need to be well-defined in order to allow for well-defined measurement scopes.

More generally, recent research in software testing questions whether state-of-the-art test evaluation criteria and tools are effective. A recent study about mutation tools in Java [153] questions their effectiveness in terms of the mutant operators used. They demonstrated a major weakness of PIT in which the default set of PIT operators generated mutants that were too easy to kill. At the same time, they suggested a list of extended mutation operators that were demonstrated to be more efficient. A recent study of EvoSuite applied in practice [86] found that even though AUTG improves coverage criteria such as code coverage and mutation scores, overall no additional real defects could be identified.

16.5 Supporting Experimentation

Over the years, researchers have identified the need to support software experimentation through testing techniques in order to improve efficiency [71]. Mutation

testing in particular, is seen as a promising way of conducting empirical assessments in testing research [7] (i.e., planting synthetic faults that tools under study have to uncover). Related work and research that supports experimentation activities in software engineering can be roughly classified into three categories —

- frameworks and guidelines,
- study corpora,
- platforms/infrastructures.

The main contribution and novelty of LASSO is that it integrates with popular experimental frameworks (including measurements), and provides experimentation guidelines as discussed in Chapter 15. These can be used by researchers to encode their individual study designs into executable analysis pipelines that output results for assessment in a classic, data-driven way using statistical tools. The activities of the operation phase of experimentation are fully automated, while LSL scripts provide high transparency of the variables controlled in the study.

Study corpora that support aspects of software experimentation typically focus on the benchmarking of tools that solve a certain engineering problem (e.g., AUTG tools). Corpora such as DEFECTS4J even provide scripts to access testing and measurement harnesses. These corpora are still limited, however, while the live data sets that can be distilled using LASSO CURATE, support the automated curation of data sets based on individual curation criteria that satisfy the needs of a particular study domain. At the same time, the data sets share the same desirable properties of being reusable by others in order to support replication studies.

The BOA platform also supports automated experimentation in terms of hypothesis testing (mostly mining tasks). Its DSL allows users to formulate and answer questions about software code elements at the level of repositories in terms of mining tasks declared in BOA's DSL [75]. Since it does not allow the definition of integrated pipelines like LSL, benchmarking studies that are typically conducted in software engineering research are not possible. As mentioned previously, the selection criteria available to users is also limited to static properties inferred from AST analyses as well as metadata from repositories. Similarly, big code data sets that offer vast amounts of version control data as discussed in Section 16.3 suffer from the same limitations.

A special type of platform (or infrastructure) that supports software experimentation are benchmarking infrastructures. In contrast to LASSO, these are typically used to support benchmarking studies of tools and techniques in a certain research field, and thus for a concrete problem domain. A recent infrastructure that was proposed for the benchmarking of AUTG tools in Java is JUGE [66]. Even though

JUGE provides an extended scripting framework to automate the benchmarking of tools, it is limited to one problem domain only (i.e., test generation) as opposed to LASSO. Similarly, other examples of benchmarking infrastructures that are limited to a specific problem domain include BIGCLONEBENCH [226] and BIGCLONEEVAL [225] for code clone detection techniques.

17 Conclusion

This final chapter summarises the core contributions made by this thesis based on the requirements formulated in Section 1.3. It then discusses the current limitations of the approach and prototype platform before considering how they could be addressed by future research. Finally, the last section presents our future visions for a LASSO user community.

17.1 Summary of Contributions

This thesis has presented a new kind of analysis platform, called a "software observatorium", that allows practitioners and researchers to describe and perform observation-based analyses of large numbers of software systems by writing abstract pipeline scripts in a dedicated DSL (i.e., LSL). This offers practitioners and researchers a practical way to exploit the notion of behaviour (i.e., semantics) as well as static properties in their analyses and studies of software systems. By automating the behaviour-aware selection, measurement, analysis and comparison of software systems on a large-scale, LASSO significantly reduces the effort traditionally involved in repository mining and validation studies, and facilitates new kinds of services and studies that were hitherto impractical. It frees practitioners and researchers from many of the tedious and time-consuming tasks involved in traditional observation-based mining and validations tasks, such as curating data sets and writing harnesses to access them.

17.1.1 Requirements

The observatorium presented in this thesis was expressly developed to support the requirements formulated in Section 1.3. In this section the core features and contributions of the thesis are discussed with respect to these requirements.

R1. Automatically Curated Software Corpora

Requirement *R1* addresses the fundamental challenge of Problem *P1* – the fact that today, executable corpora are invariably curated manually. In order to obviate the huge amounts of manual effort involved, we proposed a novel curation approach

that is fully automated and allows software systems to be curated with high precision based on rich, behaviour-aware curation criteria. The core contributions we make include support for —

- *Executable Software Corpora*: executable software corpora built on a uniform repository model and approach to transform diverse software code repositories with different layouts into a (super)corpus of executable software systems,

- *Advanced Selection Criteria*: an analysis service, LASSO CURATE, that facilitates the formulation of individual, behaviour-aware curation criteria in order to "distil" software data sets with desired properties, on demand, from the underlying corpus.

The design of the executable corpus through the "mavenisation" of software artefacts increases the likelihood that selected software systems are executable and hence testable. Moreover, the integration of well-known executable data sets including real-world repositories like Maven Central and data sets for experimentation (SF110, 50-K etc.), allows users of the observatorium to automatically curate "live data sets" of executable software systems of interest using the pipeline language LSL.

R2. True-Behaviour-Aware Software Selection and Comparison

The core contribution to the realisation of behaviour-aware selection and comparison of software systems is the arena at the heart of the observatorium. The arena implements an enhanced test-driven selection approach that is based on the concept of behaviour sampling which increases the precision of behaviour-aware selection. To this end, the observatorium implements a clear separation of concerns in order to compare exhibited behaviours and measure additional properties obtainable at runtime in a flexible way. First, candidate systems are preselected from the executable corpus based on mature text-based (NLP-driven) selection techniques (e.g., IDCS), and then their actual, "true" behaviour is observed inside the arena. The analysis of the observed behaviour can be carried out at a later time in the data analytics layer of the observatorium.

The recall of the behaviour-aware selection approach is further improved by a novel, systematic adaptation technique that attempts to synthesise adapters to resolve interface mismatches. The adaptation approach provides a best-effort strategy which prioritises the creation of adapters to tackle the core challenge of the combinatorial explosion in adaptation mappings.

R3. True-Behaviour-Aware System Boundary Models

The technology developed in this thesis makes two core contributions to addressing Problem *P3* – the fact that at the time of writing, strategies for defining system boundaries in analysis approaches are either missing, behaviour-agnostic (i.e., defined statically) or limited. The presented observatorium addresses this requirement in two basic ways.

First, it defines an efficient, scalable analysis architecture based on a novel system boundary model. This allows users to define custom system scope criteria systematically in order to determine the "extent" of software systems based on their delivered functionality. Second, scope-awareness is built into the observatorium's ecosystem. It is a first-class citizen in LSL pipelines (i.e., as part of the profile block), and it is implemented by the arena's measurement harnesses which makes measurements at specific, individual scopes. Since this measurement harness can be customised and extended, users can easily define new custom scopes.

R4. Unified Stimulus/Response Data Structure and Analysis Platform

There are many obstacles to large-scale, dynamic software analysis, including the large variety of heterogeneous and disconnected languages/tools for defining software stimuli (i.e., tests), executing multiple software systems, recording the results and extracting useful information from them. In order to resolve these obstacles, the thesis makes the following contributions —

- *sequence sheets* based on the SSN notation to formalise test sequences and to systematically record the corresponding behaviour of systems,
- *SRMs* to provide a compact stimulus/response model including a navigational model for setting up configurations of systems and sequences, and for storing execution records capable of holding analysis attributes of any types,
- the *arena* as the venue for performing multiple executions of multiple systems, and,
- the *analysis platform* that allows SRMs to be analytically processed, "online" (script-driven using LSL) as well as "offline" (data-driven in external data analytics tools).

At the time of writing, no comparable approach has proposed a unified model for large-scale software observation, nor applied such a strict separation of concerns when performing (dynamic) analysis of software. Separating the observation of behaviour from the analysis of behaviour enables deep, data-driven (offline) behavioural relationships to be identified in big code approaches.

R5. Dedicated Pipeline Definition Language

Finally, the dedicated pipeline definition language, LSL, is a novel DSL that provides users with a unified view of the whole data creation and analysis pipeline. It bundles all the capabilities of the observatorium in one place. The LSL language allows users to design and write powerful analysis pipelines based on a simple domain model in a hybrid declarative/imperative way that scales to large software repositories. Depending on the goal, LSL serves as a dynamic query language to enforce concise selection criteria (e.g., reuse criteria in LASSO SEARCH and curation criteria in LASSO CURATE). Since the observatorium is designed with extensibility in mind, users can "plug-in" new required analysis approaches (e.g., tools and techniques) thanks to the abstract "action" model design underpinning LSL.

17.1.2 Validity of Hypotheses

The validity of the three hypotheses formulated at the beginning of this thesis were explored using the design science methodology (Section 1.4). The results of our validity analyses were discussed in the demonstration and evaluation chapters (see Part V).

The construction of the prototype platform LASSO confirms Hypothesis 1 concerning the practical feasibility of the envisaged observatorium. Chapter 12 demonstrated the overall feasibility of building and designing a practical platform in Java that meets all the requirements formulated in the introduction, and implements all the approaches and concepts proposed. Using the analysis services built on top of LASSO, we were able to demonstrate that such a general-purpose, dynamic analysis platform can be both efficient (i.e., in terms of providing a distributed architecture for conducting dynamic analyses), and offer added value to users (i.e., in terms of its usability and extensibility).

In order to confirm Hypothesis 2, we designed and implemented four services (LASSO SEARCH, LASSO CURATE, LASSO TESTGEN, LASSO TESTAMP) on top of the LASSO platform, in the form of reusable and extensible LSL pipelines. These services either provide better performance than existing comparable tools (i.e., for software reuse, corpus curation and software testing) or provide novel solutions that do not exist elsewhere at the time of writing (i.e., for exploiting the diversity in repositories).

Finally, based on our extensive discussion of how the pipelines can be used to automate and transcribe abstract study designs into executable study designs, and the practical study of LASSO TESTGEN published in literature, we also confirmed the validity of Hypothesis 3. The observatorium does indeed provide a more powerful and usable platform for evaluating software engineering tools.

17.2 Limitations and Future Research Directions

Naturally, as a prototype platform, LASSO has numerous limitations. However, many of these were the result of trade-offs made for the purpose of demonstrating feasibility rather than providing optimal services or functionality for end users. In virtually all cases there are numerous ways in which the LASSO prototype implementation could be enhanced. This section discusses the main limitations and possible strategies for addressing them.

Behaviour Sampling and Dynamic Analysis

The observatorium presented in this thesis realises behaviour sampling in the form of a test-driven code search engine. Since behaviour sampling is based on dynamic analysis, it inherits dynamic analysis's property of insufficient soundness, because exhaustive testing is typically impractical. The soundness and precision of static analysis versus dynamic analysis was discussed in Section 16.1.

Like software testing, due to the aforementioned incompleteness, one of the core challenges in the behaviour-aware selection of software systems is the selection of high-quality test sequences that adequately approximate the functionality of interest (i.e., in terms of actuations). At the time of writing, practitioners who want to conduct behaviour-aware selection can only rely on general-purpose test quality criteria from the field of software testing, since no behaviour-sampling-oriented quality criteria have been proposed to date. As a consequence, the question of how many tests are enough to gain high confidence that a system implements a certain functional abstraction has still only been partially answered. In order to tackle this challenge, LASSO can be used to further explore possible quality criteria in the vein of existing related work [191, 223, 141].

Another related challenge that needs further investigation is the identification of behavioural relationships (see Section 3.4.1) between software systems and functional abstractions in SRMs. The mining of information from SRMs by comparing actuations, in particular, presents a data representation problem, since the (technical) serialisation and representation of program objects in a tabular representation (Section 6.6.1) for efficient "offline" comparison needs further improvement. Currently, all objects are serialised into a string representation that effectively transforms the problem of behaviour comparison into a string similarity problem. However, since object graphs (hierarchies) can be arbitrarily complex, in general the comparison of two strings is a non-trivial problem. As a future research direction, we believe our string serialisation schema based on JSON can be further enhanced to create comparable documents that also take into account type adaptation (i.e., by defining

aliases for common types). Moreover, by applying advanced data mining techniques (e.g., classification etc.) to SRMs, we also believe it is feasible to "abstract" from concrete stimuli and/or responses in order to create behaviour taxonomies that can be used to match equal or similar actuations (i.e., subbehaviour). Effectively, this problem is similar to the idea of adaptation, but here at the level of concrete observed outputs (i.e., objects).

Finally, in its current form, LASSO does not solve the oracle problem [24]. The observatorium and its SRM configurations (Section 11.3), however, present an opportunity to develop new ways of generating automated test oracles (e.g., by creating domain-specific oracles through the mining of actuations from SRMs at a large scale [151]).

Adaptation of Software Systems

The synthesis of adapters (Section 9.3.2), is arguably one of the key enabling technologies of behaviour sampling in modern, large-scale software repositories. It therefore comes as no surprise that the adaptation capabilities of behaviour sampling engines usually constitutes their "Achilles heel", and represent the main bottleneck in the efficient harvesting of functionally equivalent, alternative implementations of functional abstractions. State-of-the-art adaptation approaches, including ours, still face three challenges that need to be resolved in order to further improve recall, including —

- *Relevance/Efficiency*: Adaptation strategies and operations often face a combinatorial explosion in the size of the search space of possible adaptations, so identifying relevant adapters in reasonable time is still a major challenge.
- *Custom Types*: Today's wealth of custom type definitions and their complexity (i.e., inheritance, polymorphism etc.) requires more sophisticated adaptation operations that establish efficient mappings between "compatible", user-defined types.
- *Structural vs "Behavioural" Adaptation*: Adaptation strategies are usually designed to solve structural adaptation issues (i.e., interface mismatches) under the assumption that the adaptee implements the required functionality. Typically, therefore they do not consider "behavioural" mismatches (i.e., missing, superfluous or slightly different, although related, behaviour) that may also require adaptation as well.

We believe it is possible to tackle the combinatorial search space explosion in adaptation scenarios in two possible ways. First, inspired by the field of search-based

software engineering [107], one way of synthesising relevant adapters in less time is to recast the adaptation problem as an optimisation problem, allowing meta-heuristics and associated optimisation algorithms to be applied [63, 64], similar to the AUTG tool EvoSuite. Secondly, with the support of the observatorium, it is also possible to recast adaptation as a data mining problem and leverage the past "knowledge" gathered by previous LSL pipeline executions to empower machine learning algorithms to identify patterns and classifications that can help to synthesise relevant adapters in less time.

The flexibility given to developers by object-oriented languages presents a severe challenge for the realisation of efficient adaptation strategies. The custom types, in particular, are a great tool for developers to realise domain-specific structural designs and interfaces, but at the same time the idiosyncratic choices made by developers greatly increase the complexity of the adaptation problem. This thesis has proposed a systematic adaptation approach built around advanced adaptation strategies and operators. Adaptation operators are typically based on a mix of basic language properties and heuristics obtained from manual inspection of code. In order to tackle the issue of custom types, we envision the large scale use of code mining techniques to identify more general, widely applicable adaptation operators. The pairwise "interplay" of adaptation operators also needs further exploration.

Finally, based on our experience with the LASSO platform and manual inspection of real-world systems harvested from Maven Central, we identified a significant potential to harvest even more alternative implementations of functional abstractions if appropriate solutions for "behavioural adaptation" could be found that go beyond structural adaptation of the expected interface signatures. It is often the case that a system realises the fundamental behaviour of a functional abstraction, but in a slightly different way (i.e., there are one or more actuations that disagree with the desired behaviour). Since it is the task of the observatorium user (i.e., domain expert) to decide if such alternative implementation candidates are still relevant, functional adaptation may be performed at the level of mining SRMs. In order to automate adaptation to the greatest extent possible, on the other hand, behavioural adaptation may also be realised alongside structural adaptation by synthesising behavioural code. In this case, we envision a set of adaptation operators that add "behaviour" inside adapters (e.g., adding code blocks such as pre- or post-conditions), or more extremely, change the behaviour of an existing system (e.g., using code instrumentation). Finally, recent advances in the related areas of program synthesis and program optimisation may also be useful for identifying behavioural adaptations (e.g., [25, 26, 150]).

Measurement

LASSO operates on the premise that virtually any kind of tool can be integrated by defining new actions abstracting from their functionality. However, this is not always possible. Chapter 5 describes the observatorium's measurement model that supports the definition of behaviour-aware system boundaries based on specified scoping criteria. Furthermore, Chapter 12 emphasised that LASSO is built with extensibility in mind, and thus facilitates the integration of external tools and techniques.

In practice, however, although reusable measurement tools can easily be integrated into LASSO rather straightforwardly (e.g., JaCoCo, PIT and EvoSuite), the way they measure properties of software systems cannot, because they often hard-code how measurements are made. Typically, they only offer one particular scope definition and thus cannot accommodate the kind of flexible scope definitions supported by LASSO.

A practical way to tackle this challenge is to either (a) identify compatible "configurations" of the tools, (b) exploit the nature of their generated reports, (c) reuse partial components of the tools, or (d) modify the functionality of existing tools to make them compatible to LASSO's system boundary model. For example, to measure code coverage using different scoping criteria using JaCoCo, we configured it to generate fine-grained reports and re-evaluated them using custom criteria for class-level measurements and method-level measurements etc. In order to enable mutation testing on arbitrary classes and methods using PIT, we integrated its underlying mutation engine into our arena in order to have fine-grained control over the measurement process. In future research, we envision the development of more efficient ways to integrate external tooling as well as better ways to realise flexible measurement scopes.

The LASSO prototype platform offers all the ingredients needed to realise (Java) performance benchmarks, but it is not yet specifically optimised for statistically rigorous Java performance evaluations at a large scale [92]. To achieve this, many factors need to be considered, such as background noise (i.e., other resource-intensive applications running in the background can influence measurements), which may bias performance measurements such as keeping track of execution times, resource usage etc. In order to facilitate "sensible" measurements of performance metrics, we plan to extend the LASSO platform with explicit configuration options to prevent significant background noise. This will allow performance measurements to be made at the scale of big code by allowing multiple machines to make comparable measures for performance benchmarking. A first step has already been taken, since LASSO allows machines of similar computing power to be selected in order to ensure

comparable results. Furthermore, the containerisation of execution environments provides fine-grained control over available resources.

Ultimately, LASSO's flexible architecture can be exploited to add support for additional hardware architectures, hence additional execution environments. This allows users to obtain and collect additional important dynamic properties about software systems such as their energy consumption (e.g., [113]) at a large scale (e.g., by either using dedicated machines, virtualisation or emulators).

Software Repositories

We have successfully demonstrated the creation of a large scale artefact repository which integrates numerous popular software engineering corpora in Section 12.4. The unified repository model underpinning LASSO attempts to simplify the integration of heterogeneous software repositories, while increasing the likelihood that the software systems therein can be executed. In practice, however, there are still many technical problems (e.g., missing build information) that impede the "automated" execution of arbitrary software systems harvested from software repositories. Future research, therefore, needs to develop more robust, automated build script synthesis strategies to further increase automatic executability, thereby reducing the need for manual effort. In the long-term, we plan to integrate more data sources, including additional software engineering corpora and popular large-scale repositories.

Software system diversity is an interesting property that we attempted to exploit in our software testing applications (LASSO TESTGEN and LASSO TESTAMP). Even though we were able to demonstrate that diversity can help improve the quality of tests (cf. Section 15.3), to the best of our knowledge there is still no proposed, scalable way to measure implementation diversity at the repository level effectively. As a future research direction, we envision another analysis service built on top of the LASSO platform that is able to measure diversity similar to approaches such as Carzaniga et al.'s [47].

Analysis Services

The modularity of LSL pipelines in terms of the action model used offers the opportunity to seamlessly integrate with external platforms and tools that (a) provide sophisticated (static) code analysis capabilities such as the monitoring of code quality (e.g., SONARGRAPH [121] and SONARQUBE) [216], or (b) give access to data sources using custom query languages (e.g., BOA [74] or CODEDJ [166]). We plan to integrate these services in future iterations of the LASSO prototype.

Recent advances in deep models for code (AI) like CODEX (based on GPT-3) have demonstrated great potential to synthesise software based on natural language queries [49]. Since these code models are limited to static techniques, and are typically trained to generate code based on the code found in large software repositories (e.g., GitHub), they do not necessarily generate high quality code (i.e., may even perpetuate bad coding habits or undesired code vulnerabilities). LASSO's capabilities, however, can be used to create more and better training data for code models, and conduct better evaluations of their effectiveness.

From an academic perspective, the versatility of LASSO's analysis capabilities give lecturers and teachers the opportunity to automate the grading process of work assignments and exams in programming courses by assembling custom analysis pipelines. In this case, we envision that students submit their work to a repository (e.g., managed by Git) which is then continuously crawled and indexed by LASSO's underlying corpus. The test-driven selection capability of the platform can be used to automatically assess the behaviour of the solutions provided by the students with a set of test sequences, on the one hand. Second, the additional analyses provided by the observatorium can be used to set up an analysis pipeline for plagiarism checks (e.g., using clone detection), or for monitoring and comparing code quality (e.g., measuring metrics). Finally, custom LSL actions may be defined for scoring purposes.

Towards a LASSO Community

LASSO has been designed to allow users to define and execute "big" software studies in an abstract way with minimum manual effort. To this end, the architecture of LASSO has been designed from the ground up to maximise "extensibility", "shareability" and "integratability" with other specialised tools. LASSO has therefore been released under a friendly Open Source license and its project sources are accessible and managed in a Git repository[1]. Contributions from the wider software engineering research community are therefore welcome.

A hosted instance of LASSO is available over the Internet, as-a-service, and private versions of LASSO can be installed in users' local computing environments.

To help users learn how to use LASSO, sample data is also available in the form of example LSL scripts and study results for demonstration purposes. In the long term, if LASSO is used and a community grows, we plan to extend the platform with more support for cooperation and interaction. To this end, we plan to set up a web-based "repository" in which LASSO users can share their LSL scripts and workspaces (i.e., study results) to encourage replication studies. This will be achieved by making the current LASSO web front-end accessible to a larger user base.

[1] see `https://softwareobservatorium.github.io/`

Appendix

LSL Scripts

A

A.1 Analysis Services Built with LASSO

A.1.1 LASSO Search

Test-Driven Search

```
dataSource 'mavenCentral2020'
def interfaceSpec = '''Stack {
                push(Object)->Object
                pop()->Object
                peek()->Object
                size()->int }'''
study(name:'Stack-TDS') {
    action(name:'select', type:'Select') {
        abstraction('Stack') { // interface-driven code search
            queryForClasses interfaceSpec
            rows = 10
            excludeClassesByKeywords(['private', 'abstract'])
            excludeTestClasses()
            excludeInternalPkgs()
            filter 'complexity:[1 TO *]'
        }
    }

    action(name: 'clonesAlt', type: 'Nicad6') { // reject code clones
        cloneType = "type2"
        collapseClones = true

        dependsOn 'select'
        includeAbstractions 'Stack'
        profile {
            environment('nicad') {
                image = 'nicad:6.2'
            }
        }
    }

    action(name:'filter',type:'ArenaExecute') { // test filter
        sequences = [
                // parameterised sheet (SSN) with default input parameter values
                // expected values are given in first row (oracle)
                'pushPop': sheet(p1:'Stack', p2:5) {
                    row  '',    'create', '?p1'
                    row '?p2',  'push',   'A1',     '?p2'
```

```
                    row '?p2',  'peek',   'A1'
                    row     1,  'size',   'A1'
                    row '?p2',  'pop',    'A1'
                    row     0,  'size',   'A1'
                }
        ]
        maxAdaptations = 1 // how many adaptations to try

        dependsOn 'select'
        includeAbstractions 'Stack'
        profile('myTdsProfile') {
            scope('class') { type = 'class' }
            environment('java8') {
                image = 'maven:3.5.4-jdk-8-alpine'
            }
        }

        whenAbstractionsReady() {
            def stack = abstractions['Stack']
            def expectedBehaviour = toOracle(srm(abstraction: stack).sequences)
            // returns a filtered SRM
            def matchesSrm = srm(abstraction: stack)
                    .systems // select all systems
                    .equalTo(expectedBehaviour) // functionally equivalent

            // continue pipeline with matched systems only
            stack.systems = matchesSrm.systems
        }
    }

    action(name:'rank', type:'Rank') { // rank based on two criteria
        strategy = 'HDS_SMOOP' // SOCORA ranking strategy
        criteria = ['IndexMeasurements.m_static_loc_td:MIN:1',
                    'cc.branch.total:MIN:2']

        dependsOn 'filter'
        includeAbstractions '*'
    }
}
```

List. 22: LASSO SEARCH - Test-Driven Search Pipeline in LSL

Code-Driven Search

```
dataSource 'mavenCentral2020'

study(name:'Stack-CDS') {

    action(name:'selectStack', type:'Select') {
        abstraction('Stack') { // select single, known class
            queryForClasses '*:*'
            filter 'id:"4e73bb0d-f01f-43e5-bf46-7ab7870a289f"' // known class
        }
    }

    action(name:'executeRef',type:'EvoSuite') { // generate tests
        searchBudget = 120

        dependsOn 'selectStack'
        includeAbstractions 'Stack'
        profile {
            environment('java8') {
                image = 'maven:3.5.4-jdk-8'
            }
        }
    }

    action(name: 'selectAlt', type: 'Select') { // select alternative impls.
        dependsOn 'executeRef'
        includeAbstractions 'Stack'

        execute() {
            List refImpls = abstractions['Stack'].implementations
            refImpls.each { impl ->
                abstraction(impl) {
                    queryByExample impl, 'class'
                    rows = 10

                    excludeClassesByKeywords(["private", "abstract"])
                    excludeTestClasses()
                    excludeInternalPkgs()

                    excludeImplementation(impl.id)
                }
            }
        }
    }

    action(name: 'clonesAlt', type: 'Nicad6') { // reject code clones
        cloneType = "type2"
        collapseClones = true
        refActionRef = "executeRef"

        dependsOn 'selectAlt'
        includeAbstractions '*-*'
        profile {
```

```
            environment('nicad') {
                image = 'nicad:6.2'
            }
        }
    }

    action(name:'arena',type:'ArenaExecute') { // execute in the arena
        disablePartitioning = true
        maxPermutations = 1
        task = 'Amplify'
        features = ['mutation', 'cc'] // measure MS and BC

        dependsOn 'clonesAlt'
        includeAbstractions '*-*'
        includeSequences '*' // take any
        profile {
            environment('java8') {
                image = 'openjdk:8-jdk-alpine'
            }
        }

        whenAbstractionsReady() {
            // determine functionally equivalent ones
            ...
        }
    }
}
```

List. 23: LASSO SEARCH - Code-Driven Search Pipeline in LSL

A.1.2 LASSO Curate

Behaviour-Aware Curation Criteria

```
dataSource 'mavenCentral2020'

study(name:'Behaviour-Aware-Curation-Criteria') {

    action(name:'selectRandom', type:'Select') { // random sampling
        abstraction('Random') {
            queryForClasses '*:*'
            random = true // random selection
            rows = 1000 // number of classes to select
        }
    }

    action(name:'executeRef',type:'EvoSuite') { // generate tests
        searchBudget = 120

        dependsOn 'selectRandom'
        includeAbstractions 'Random'
        profile {
            environment('java8') {
                image = 'maven:3.5.4-jdk-8'
            }
        }
    }

    action(name: 'selectAlt', type: 'Select') { // select alternative impls.
        dependsOn 'executeRef'
        includeAbstractions 'Random'

        execute() {
            List refImpls = abstractions['Random'].implementations
            refImpls.each { impl ->
                abstraction(impl) {
                    queryByExample impl, 'class'
                    rows = ...

                    excludeClassesByKeywords(["private", "abstract"])
                    excludeTestClasses()
                    excludeInternalPkgs()

                    excludeImplementation(impl.id)
                }
            }
        }
    }

    action(name: 'clonesAlt', type: 'Nicad6') { // reject code clones
        cloneType = "type2"
        collapseClones = true
        refActionRef = "executeRef"

```

```
51            dependsOn 'selectAlt'
52            includeAbstractions '*-*'
53            profile {
54                environment('nicad') {
55                    image = 'nicad:6.2'
56                }
57            }
58        }
59
60        ...
61    }
```

List. 24: LASSO CURATE - Behaviour-Aware Curation Criteria in LSL

Behaviour-Agnostic Curation Criteria

```
dataSource 'mavenCentral2020'

study(name:'Behaviour-Agnostic-Curation-Criteria') {

    action(name:'selectRandom', type:'Select') { // random sampling
        abstraction('Random') {
            queryForClasses '*:*'
            random = true // random selection
            rows = 1000 // number of classes to select
        }
    }

    action(name:'executeRef',type:'EvoSuite') { // generate tests
        searchBudget = 120

        dependsOn 'selectRandom'
        includeAbstractions 'Random'
        profile {
            environment('java8') {
                image = 'maven:3.5.4-jdk-8'
            }
        }
    }

    action(name: 'clones', type: 'Nicad6') { // reject code clones
        cloneType = "type2"
        collapseClones = true

        dependsOn 'executeRef'
        includeAbstractions '*-*'
        profile {
            environment('nicad') {
                image = 'nicad:6.2'
            }
        }
    }

    ...
}
```

List. 25: LASSO CURATE - Behaviour-Agnostic Curation Criteria in LSL

A.1.3 LASSO TestGen

```
dataSource 'mavenCentral2020'

def altImpls = 10
def adapterImplementations = 1
def refEvoSuiteTimeBudget = 120
def altEvoSuiteTimeBudget = 120

study(name:'LASSO-TestGen') {
    action(name:'select', type:'Select') {
        abstraction('Stack') { // assume known CUT
            queryForClasses '*:*'
            filter 'id:"4e73bb0d-f01f-43e5-bf46-7ab7870a289f"' // known class
        }
    }

    action(name: 'selectAlt', type: 'Select') { // select alternative impls.
        dependsOn 'select'
        includeAbstractions 'Stack'

        execute() {
            List refImpls = abstractions['Stack'].implementations
            refImpls.each { impl ->
                abstraction(impl) {
                    queryByExample impl, 'class'
                    rows = altImpls
                    excludeClassesByKeywords(["private", "abstract"])
                    excludeTestClasses()
                    excludeInternalPkgs()
                    excludeImplementation(impl.id)
                }
            }
        }
    }

    action(name: 'clonesAlt', type: 'Nicad6') { // reject code clones
        cloneType = "type2"
        collapseClones = true
        refActionRef = "select"

        dependsOn 'selectAlt'
        includeAbstractions '*-*'
        profile {
            environment('nicad') {
                image = 'nicad:6.2'
            }
        }
    }

    profile('evosuite') { // execution profile
        scope('class') { type = 'class' }
        environment('java8') {
            image = 'maven:3.5.4-jdk-8'
```

```
53              }
54          }
55
56          action(name:"evosuiteRef",type:'EvoSuite') { // generate tests for reference impl.
57              ignoreMissingReport = true
58              searchBudget = refEvoSuiteTimeBudget
59
60              dependsOn 'select'
61              includeAbstractions 'Stack'
62              profile('evosuite')
63          }
64
65          action(name:"evosuiteAlt",type:'EvoSuite') { // generate tests for alternative impls.
66              ignoreMissingReport = true
67              searchBudget = altEvoSuiteTimeBudget
68
69              dependsOn "clonesAlt"
70              includeAbstractions '*-*'
71              profile('evosuite')
72          }
73
74          action(name:"arena",type:'Arena') { // obtain test sequences
75              maxPermutations = adapterImplementations
76              task = 'Amplify'
77              referenceImplementationOnly = true
78
79              dependsOn "evosuiteAlt"
80              includeAbstractions '*-*'
81              includeSequences '*'
82              profile('evosuite')
83          }
84      }
```

List. 26: LASSO TESTGEN - Diversity-Driven Test Generation Pipeline in LSL

A.1.4 LASSO TestAmp

```
dataSource 'mavenCentral2020'
def interfaceSpec = '''Stack {
                push(Object)->Object
                pop()->Object
                peek()->Object
                size()->int }'''
study(name:'LASSO-TestAmp') {
    action(name:'select', type:'Select') {
        abstraction('Stack') { // select 10 stack classes based on IDCS
            queryForClasses interfaceSpec
            rows = 10
            excludeClassesByKeywords(['private', 'abstract'])
            excludeTestClasses()
            excludeInternalPkgs()
            filter 'complexity:[1 TO *]'
        }
    }

    action(name: 'clonesAlt', type: 'Nicad6') { // reject code clones
        cloneType = "type2"
        collapseClones = true

        dependsOn 'select'
        includeAbstractions 'Stack'
        profile {
            environment('nicad') {
                image = 'nicad:6.2'
            }
        }
    }

    profile('arena') { // execution profile
        scope('class') { type = 'class' }
        environment('java8') {
            image = 'maven:3.5.4-jdk-8'
        }
    }

    action(name:'filter',type:'ArenaExecute') { // determine functionally equivalent
    ↪  classes
        sequences = [
                'pushPop': sheet(p1:'Stack', p2:5) {
                    row  '',    'create', '?p1'
                    row '?p2',  'push',   'A1',     '?p2'
                    row '?p2',  'peek',   'A1'
                    row     1,  'size',   'A1'
                    row '?p2',  'pop',    'A1'
                    row     0,  'size',   'A1'
                }
        ]
        maxAdaptations = 1

```

```
52          dependsOn 'select'
53          includeAbstractions 'Stack'
54          profile('arena')
55
56          whenAbstractionsReady() {
57              def stack = abstractions['Stack']
58              def expectedBehaviour = toOracle(srm(abstraction: stack).sequences)
59              // returns a filtered SRM
60              def matchesSrm = srm(abstraction: stack)
61                      .systems // select all systems
62                      .equalTo(expectedBehaviour) // functionally equivalent
63              // continue pipeline with matched systems only
64              stack.systems = matchesSrm.systems
65          }
66      }
67
68      action(name:'evosuite',type:'EvoSuite') { // generate tests
69          ignoreMissingReport = true
70          searchBudget = 120
71
72          dependsOn 'filter' // mandatory
73          includeAbstractions 'Stack'
74          profile('arena')
75      }
76
77      action(name:"arena",type:'Arena') { // amplify test sequences
78          disablePartitioning = true
79          maxPermutations = 1
80          task = 'Amplify'
81          exportCsv = true
82
83          dependsOn "evosuite"
84          includeAbstractions 'Stack'
85          includeSequences '*'
86          profile('arena')
87      }
88  }
```

List. 27: LASSO TestAmp - Diversity-Driven Test Amplification Pipeline in LSL

A.2 Study Design Realised with LASSO

A.2.1 LASSO TestGen

Part I - TestGen

```
dataSource 'mavenCentral2020'

def totalNoOfRandomClasses = 10
def altImpls = 10
def adapterImplementations = 1 // how many adapters to try
def refEvoSuiteTimeBudget = 120
def altEvoSuiteTimeBudget = 120

def studyRepetitions = 10

study(name:'TestGen-Study') {

    action(name: 'selectRandom', type: 'Select') { // random sampling
        abstraction('Random') {
            queryForClasses '*:*'
            rows = totalNoOfRandomClasses
            random = true
            // ...
        }
    }

    action(name: "selectAlt", type: 'Select') { // select alternative impls.
        dependsOn "selectRandom"
        includeAbstractions 'Random'

        execute() {
            List refImpls = abstractions['Random'].implementations
            refImpls.each { impl ->
                abstraction(impl) { // by example
                    queryByExample impl, 'class', false
                    rows = altImpls
                    // ...
                }
            }
        }
    }

    action(name: "clonesAlt", type: 'Nicad6') { // reject code clones
        cloneType = "type2"
        collapseClones = true
        refActionRef = "selectRandom"

        dependsOn "selectAlt"
        includeAbstractions '*-*'
        profile {
            environment('nicad') {
                image = 'nicad:6.2'
```

```
48              }
49          }
50      }
51
52      // repetitions
53      for(int repetition = 0; repetition < studyRepetitions; repetition++) { // repeat
54          action(name:"evosuiteRef_${repetition}",type:'Evosuite') { // generate tests for
        ↪ reference impl.
55              ignoreMissingReport = true
56              searchBudget = refEvoSuiteTimeBudget
57              stoppingCondition = "MaxTime"
58              criterion =
                ↪ "LINE:BRANCH:EXCEPTION:WEAKMUTATION:OUTPUT:METHOD:METHODNOEXCEPTION:CBRANCH"
59
60              // kill process after searchBudget * timeoutMultiplier
61              timeoutMultiplier = 3
62
63              dependsOn 'selectRandom' // mandatory
64              includeAbstractions 'Random'
65              includeImplementations {abName ->
66                  // check if alts exist, if not remove ref
67                  abstractions[abName].implementations.removeAll {impl ->
68                      // alts empty
69                      !actions["clonesAlt"].abstractions[impl.id]?.implementations
70                  }
71
72                  abstractions[abName].implementations
73              }
74              profile {
75                  environment('java8') {
76                      image = 'maven:3.5.4-jdk-8'
77                  }
78              }
79          }
80
81          action(name:"pitestOriginal_${repetition}",type:'Pitest') { // measure MS for
        ↪ reference impl.
82              dropFailed = true // drop if we cannot measure PIT
83
84              dependsOn "evosuiteRef_${repetition}"
85              includeAbstractions 'Random'
86              includeSequences '*' // include original tests
87              profile {
88                  environment('java8') {
89                      image = 'maven:3.5.4-jdk-8-alpine'
90                  }
91              }
92          }
93
94          action(name:"jacocoOriginal_${repetition}",type:'JaCoCo') { // measure BC for
        ↪ reference impl.
95              dropFailed = false
96
97              minimumTestCoverage = 0d
98              generateReport = false
```

```
99
100             // helper variable (scoping issues)
101             def currentRepetition = repetition
102
103             dependsOn "evosuiteRef_${repetition}"
104             includeAbstractions 'Random'
105             includeSequences '*' // include original tests
106             includeImplementations {abName ->
107                 String actionRef = "pitestOriginal_${currentRepetition}".toString()
108                 actions[actionRef].abstractions[abName].implementations // only run those
                    ↪ which passed PIT
109             }
110             profile {
111                 environment('java8') {
112                     image = 'maven:3.5.4-jdk-8-alpine'
113                 }
114             }
115         }
116
117         action(name:"evosuiteAlt_${repetition}",type:'Evosuite') { // generate tests for
            ↪ alternative impls.
118             // configuration
119             ignoreMissingReport = true
120             searchBudget = altEvoSuiteTimeBudget
121             stoppingCondition = "MaxTime"
122             criterion =
                ↪ "LINE:BRANCH:EXCEPTION:WEAKMUTATION:OUTPUT:METHOD:METHODNOEXCEPTION:CBRANCH"
123
124             stopAfter = stopAfterClasses // stop after X successful classes
125
126             // kill process after searchBudget * timeoutMultiplier
127             timeoutMultiplier = 3
128
129             // helper variable (scoping issues)
130             def currentRepetition = repetition
131
132             dependsOn "clonesAlt"
133             includeAbstractions '*-*'
134             includeImplementations {abName ->
135                 // only return alt impls if ref exists
136                 String actionRef = "pitestOriginal_${currentRepetition}".toString()
137                 if(actions[actionRef].abstractions['Random'].implementations?.find { it.id
                    ↪ == abName }) {
138                     return abstractions[abName].implementations
139                 } else {
140                     return []
141                 }
142             }
143             profile {
144                 environment('java8') {
145                     image = 'maven:3.5.4-jdk-8'
146                 }
147             }
148
149             whenAbstractionsReady() {
```

```
150                 Map abs = abstractions as Map
151
152                 // get down to stopAfterClasses and add reference implementation.
153                 abs.each{ abName, abstraction ->
154                     abstraction.implementations =
155                             abstraction.implementations.take(stopAfterClasses) // take up
                                ↪  to 'stopAfterClasses'
156
157                     // add ref impl
158                     // only if at least one implementation
159                     if(abstraction.implementations) {
160                         String refAction = "evosuiteRef_${currentRepetition}".toString()
161                         def refImpl =
                            ↪  actions[refAction].abstractions['Random'].implementations?.find
                            ↪  { it.id == abName }
162                         if(refImpl) {
163                             abstraction.implementations.add(refImpl)
164                         }
165                     }
166                 }
167             }
168         }
169
170         action(name:"arena_${repetition}",type:'Arena') { // obtain test sequences
171             disablePartitioning = true
172             maxPermutations = adapterImplementations // this assumes that we try only the
                ↪  "best match"
173             task = 'Amplify'
174             exportCsv = true
175
176             containerTimeout = 1 * 60 * 60 * 1000L // 1hour
177
178             referenceImplementationOnly = true // required
179
180             dependsOn "evosuiteAlt_${repetition}"
181             includeAbstractions '*-*'
182             includeImplementations {abName ->
183                 // reference implementations only
184                 if(abstractions[abName].implementations?.size < 2) {
185                     return [] // don't execute this action if no alts
186                 }
187
188                 abstractions[abName].implementations
189             }
190             includeSequences '*' // take any
191             profile {
192                 environment('java8') {
193                     image = 'openjdk:8-jdk-alpine'
194                 }
195             }
196         }
197
198         action(name: "merge_${repetition}") { // merge reference impls. (plain action)
199             dependsOn "arena_${repetition}"
200             includeAbstractions '*-*'
```

```
201
202             execute {
203                 Map abs = abstractions as Map
204                 List refs = abs.collect { name, ab ->
205                     ab.implementations?.find { it.id == name }
206                 }.findAll { it != null}
207
208                 // create new abstraction
209                 def amplifiedAbstraction = abstraction(refs, "Amplified")
210
211                 log("abstraction ${amplifiedAbstraction.name} has
                    ↪  ${amplifiedAbstraction.implementations.size()} implementations")
212             }
213         }
214
215
216         action(name:"pitestAmplify_${repetition}",type:'Pitest') { // measure MS for
            ↪  generated set of tests for reference impl.
217             dropFailed = false
218
219             dependsOn "merge_${repetition}" // mandatory
220             includeAbstractions 'Amplified'
221             includeSequences '*' // include all
222             profile {
223                 environment('java8') {
224                     image = 'maven:3.5.4-jdk-8-alpine'
225                 }
226             }
227         }
228
229         action(name:"jacocoAmplify_${repetition}",type:'JaCoCo') { // measure BC for
            ↪  generated set of tests for reference impl.
230             dropFailed = false
231
232             minimumTestCoverage = 0d
233             generateReport = false
234
235             dependsOn "merge_${repetition}" // mandatory
236             includeAbstractions 'Amplified'
237             includeSequences '*' // include all
238             profile {
239                 environment('java8') {
240                     image = 'maven:3.5.4-jdk-8-alpine'
241                 }
242             }
243         }
244     }
245 }
```

List. 28: Study Design for LASSO TestGen: Part I (Study Objects $TestGen_{2n}$ and $MonoGen_2$)

```
dataSource 'mavenCentral2020'

def refEvoSuiteTimeBudget = 120
def studyRepetitions = 10
def rerunId = '981bbcbb-483b-4593-9892-68885db93934' // fetch implementations from previous study run (i.e., Part I)

study(name:'MonoGen2n-Study') {
    for(int repetition = 0; repetition < studyRepetitions; repetition++) {
        action(name:"evoTime_${repetition}",type:'Evosuite') { // generate tests
            configure {
                ignoreMissingReport = true
                searchBudget = refEvoSuiteTimeBudget // we need this as upper bound for timeouts
                stoppingCondition = "MaxTime"
                criterion =
                "LINE:BRANCH:EXCEPTION:WEAKMUTATION:OUTPUT:METHOD:METHODNOEXCEPTION:CBRANCH"

                // get no of alt. implementations by abstraction (i.e impls size * default budget)
                List refImpls = abstractions['Amplified'].implementations as List
                timeBudgetProviderByImpl = refImpls.collectEntries { impl ->
                    List allImpls = abstractions[impl.id].implementations as List
                    if(allImpls /*&& allImpls.size() > 1*/) { // non-empty
                        return [impl.id, allImpls.size() * refEvoSuiteTimeBudget] // include ref impl
                    } else {
                        return [impl.id, 1 * refEvoSuiteTimeBudget]
                    }
                }

                timeBudgetProviderDefault = 120 // fallback for missing implementation

                // KILL AFTER MAX 15 * 120 + X
                timeoutClientProcess = 20 * 120 // 40 minutes
            }

            execute {
                Map abs = abstractions
                List keys = new ArrayList(abs.keySet())
                List refs = keys.findAll { it.contains('-')}
                refs.each {
                    log("clearing abstraction '${it}'")
                    abs[it]?.implementations?.clear() // clear
                    log("size abstraction '${it}' = '${abs[it]?.implementations?.size()}'")
                }
            }

            dependsOn "${rerunId}:merge_${repetition}".toString() // fetch existing implementations from previous run based on URI scheme
            includeAbstractions '*' // Amplified all*
            includeSequences 'NONE' // include NONE (dummy placeholder, means EXCLUDE any)
```

```
47              profile {
48                  environment('java8') {
49                      image = 'maven:3.5.4-jdk-8'
50                  }
51              }
52          }
53
54          action(name:"pitestEvoTime_${repetition}",type:'Pitest') { // measure MS
55              dropFailed = false
56
57              dependsOn "evoTime_${repetition}" // mandatory
58              includeAbstractions 'Amplified'
59              includeSequences '*' // include all
60              profile {
61                  environment('java8') {
62                      image = 'maven:3.5.4-jdk-8-alpine'
63                  }
64              }
65          }
66
67          action(name:"jacocoEvoTime_${repetition}",type:'JaCoCo') { // measure BC
68              dropFailed = false
69
70              minimumTestCoverage = 0d
71              generateReport = false
72
73              dependsOn "evoTime_${repetition}" // mandatory
74              includeAbstractions 'Amplified'
75              includeSequences '*' // include all
76              profile {
77                  environment('java8') {
78                      image = 'maven:3.5.4-jdk-8-alpine'
79                  }
80              }
81          }
82      }
83  }
```

List. 29: Study Design for LASSO TESTGEN: Part II (Study Object $MonoGen_{2n}$)

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006 (cit. on pp. 62, 100).

[2] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," *IEEE Access*, vol. 7, pp. 86 121–86 144, 2019 (cit. on p. 255).

[3] N. M. Albunian, "Diversity in search-based unit test suite generation," in *Search Based Software Engineering*, T. Menzies and J. Petke, Eds., Cham: Springer International Publishing, 2017, pp. 183–189 (cit. on p. 203).

[4] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, 2018 (cit. on pp. 3, 93, 106, 118, 202).

[5] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, 1–19, 1970 (cit. on p. 63).

[6] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016 (cit. on pp. 7, 41, 59, 236, 260).

[7] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" In *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05, St. Louis, MO, USA: Association for Computing Machinery, 2005, 402–411 (cit. on p. 262).

[8] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006 (cit. on p. 260).

[9] Android Developers. "Activity." (2022), [Online]. Available: `https://developer.android.com/reference/android/app/Activity` (visited on May 1, 2022) (cit. on p. 194).

[10] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014 (cit. on pp. 223, 226, 231, 235).

[11] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *Search Based Software Engineering*, M. B. Cohen and M. Ó Cinnéide, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 33–47 (cit. on p. 202).

[12] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *Search Based Software Engineering*, M. B. Cohen and M. Ó Cinnéide, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 33–47 (cit. on p. 216).

[13] C. Atkinson, F. Barth, and D. Brenner, "Software testing using test sheets," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010, pp. 454–459 (cit. on p. 246).

[14] C. Atkinson, D. Brenner, G. Falcone, and M. Juhasz, "Specifying high-assurance services," *Computer*, vol. 41, no. 8, pp. 64–71, 2008 (cit. on pp. 53, 54, 246).

[15] C. Atkinson, M. Kessel, and M. Schumacher, "On the synergy between search-based and search-driven software engineering," in *Search Based Software Engineering*, G. Ruhe and Y. Zhang, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 239–244 (cit. on pp. 12, 203).

[16] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on software engineering*, no. 12, pp. 1491–1501, 1985 (cit. on pp. 203, 234, 256).

[17] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Science of Computer Programming*, vol. 79, pp. 241–259, 2014, Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010) (cit. on pp. 6, 106, 252, 256).

[18] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 50:1–50:39, May 2018 (cit. on p. 252).

[19] T. Ball, "The concept of dynamic analysis," in *Software Engineering—ESEC/FSE'99*, Springer, 1999, pp. 216–234 (cit. on p. 243).

[20] S. Baltes and P. Ralph, "Sampling in software engineering research: A critical review and guidelines," *Empirical Software Engineering*, vol. 27, no. 4, pp. 1–31, 2022 (cit. on pp. 199, 220, 230, 239).

[21] M. Balzer, O. Deussen, and C. Lewerentz, "Voronoi treemaps for the visualization of software metrics," in *Proceedings of the 2005 ACM Symposium on Software Visualization*, ser. SoftVis '05, St. Louis, Missouri: Association for Computing Machinery, 2005, 165–172 (cit. on p. 254).

[22] M. Balzer, A. Noack, O. Deussen, and C. Lewerentz, "Software Landscapes: Visualizing the Structure of Large Software Systems," in *Eurographics / IEEE VGTC Symposium on Visualization*, O. Deussen, C. Hansen, D. Keim, and D. Saupe, Eds., The Eurographics Association, 2004 (cit. on p. 254).

[23] B. Barns and T. Bollinger, "Making reuse cost-effective," *IEEE Software*, vol. 8, no. 1, pp. 13–24, 1991 (cit. on p. 193).

[24] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015 (cit. on pp. 31, 93, 115, 247, 260, 270).

[25] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, Hong Kong, China: Association for Computing Machinery, 2014, pp. 306–317 (cit. on p. 271).

[26] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, Baltimore, MD, USA: Association for Computing Machinery, 2015, pp. 257–269 (cit. on p. 271).

[27] J. L. Barros-Justo, F. Pinciroli, S. Matalonga, and N. Martínez-Araujo, "What software reuse benefits have been transferred to the industry? a systematic mapping study," *Information and Software Technology*, vol. 103, pp. 1–21, 2018 (cit. on p. 193).

[28] V. R. Basili, "Goal question metric paradigm," *Encyclopedia of software engineering*, pp. 528–532, 1994 (cit. on pp. 68, 216).

[29] V. R. Basili, R. W. Selby, and D. H. Hutchens, "Experimentation in software engineering," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 7, pp. 733–743, 1986 (cit. on p. 213).

[30] V. Bauer, J. Eckhardt, B. Hauptmann, and M. Klimek, "An exploratory study on reuse at google," in *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices*, ser. SER&IPs 2014, Hyderabad, India: Association for Computing Machinery, 2014, 14–23 (cit. on p. 185).

[31] V. Bauer and A. Vetro', "Comparing reuse practices in two large software-producing companies," *Journal of Systems and Software*, vol. 117, pp. 545 –582, 2016 (cit. on pp. 185, 186, 193).

[32] J. Beck and D. Eichmann, "Program and interface slicing for reverse engineering," in *Proceedings of 1993 15th International Conference on Software Engineering*, 1993, pp. 509–518 (cit. on p. 249).

[33] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003 (cit. on pp. 45, 113, 210).

[34] S. Becker, A. Brogi, I. Gorton, *et al.*, "Towards an engineering approach to component adaptation," Springer, 2006, pp. 193–215 (cit. on p. 252).

[35] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *IEEE/ACM MSR'17*, 2017, pp. 447–450 (cit. on p. 258).

[36] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007 (cit. on p. 255).

[37] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, and P. Grünbacher, *Value-based software engineering*. Springer Science & Business Media, 2006 (cit. on p. 63).

[38] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 241–250 (cit. on pp. 64, 244).

[39] C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, Jan. 2015 (cit. on p. 174).

[40] J. Bowring, A. Orso, and M. J. Harrold, "Monitoring deployed software using software tomography," in *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '02, Charleston, South Carolina, USA: Association for Computing Machinery, 2002, 2–9 (cit. on p. 253).

[41] A. Bracciali, A. Brogi, and C. Canal, "A formal approach to component adaptation," *Journal of Systems and Software*, vol. 74, no. 1, pp. 45 –54, 2005, Automated Component-Based Software Engineering (cit. on p. 252).

[42] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC Editor, RFC 8259, 2017 (cit. on p. 86).

[43] C. Canal, J. M. Murillo, P. Poizat, *et al.*, "Software adaptation.," *L'objet*, vol. 12, no. 1, pp. 9–31, 2006 (cit. on p. 252).

[44] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Comput. Surv.*, vol. 17, no. 4, 471–523, 1985 (cit. on p. 33).

[45] C. Carpineto and G. Romano, "A survey of automatic query expansion in information retrieval," *ACM Comput. Surv.*, vol. 44, no. 1, 2012 (cit. on pp. 109, 251).

[46] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè, "Cross-checking oracles from intrinsic software redundancy," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: ACM, 2014, pp. 931–942 (cit. on p. 256).

[47] A. Carzaniga, A. Mattavelli, and M. Pezzè, "Measuring software redundancy," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, Florence, Italy: IEEE Press, 2015, pp. 156–166 (cit. on p. 273).

[48] J. Caserta and R. Kimball, *The Data Warehouseetl Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. Wiley, 2013 (cit. on pp. 82, 83).

[49] M. Chen, J. Tworek, H. Jun, *et al.*, *Evaluating large language models trained on code*, 2021 (cit. on pp. 251, 274).

[50] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994 (cit. on p. 63).

[51] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1027–1040 (cit. on p. 245).

[52] H. Coles. "PIT - Real world mutation testing." (2022), [Online]. Available: `https://pitest.org/` (visited on May 1, 2022) (cit. on p. 17).

[53] W. J. Conover, *Practical nonparametric statistics*. john wiley & sons, 1999, vol. 350 (cit. on p. 226).

[54] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 219–220 (cit. on pp. 25, 146, 158, 180).

[55] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009 (cit. on pp. 244, 245).

[56] P. Cousot, "Abstract interpretation," *ACM Computing Surveys (CSUR)*, vol. 28, no. 2, pp. 324–328, 1996 (cit. on p. 4).

[57] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, 19:1–19:35, Sep. 2011 (cit. on p. 252).

[58] B. Danglot, O. Vera-Perez, Z. Yu, *et al.*, "A snowballing literature study on test amplification," *Journal of Systems and Software*, vol. 157, p. 110 398, 2019 (cit. on pp. 207, 259).

[59] B. Danglot, O. L. Vera-Pérez, B. Baudry, and M. Monperrus, "Automatic test improvement with dspot: A study with ten mature open-source projects," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2603–2635, 2019 (cit. on p. 207).

[60] A. Davoudian, L. Chen, and M. Liu, "A survey on nosql stores," *ACM Comput. Surv.*, vol. 51, no. 2, 2018 (cit. on p. 129).

[61] A. C. De Paula, E. Guerra, C. V. Lopes, H. Sajnani, and O. A. Lazzarini Lemos, "An exploratory study of interface redundancy in code repositories," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016, pp. 107–116 (cit. on pp. 107, 114).

[62] F. Dearle, *Groovy for Domain-Specific Languages*. Packt Publishing, 2010, vol. 10 (cit. on pp. 129, 142).

[63] K. Deb, "Multi-objective optimisation using evolutionary algorithms: An introduction," in *Multi-objective evolutionary optimisation for product design and manufacturing*, Springer, 2011, pp. 3–34 (cit. on p. 271).

[64] K. Deb, "Multi-objective optimization," in *Search methodologies*, Springer, 2014, pp. 403–449 (cit. on p. 271).

[65] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978 (cit. on p. 261).

[66] X. Devroey, A. Gambi, J. P. Galeotti, *et al.*, "JUGE: an infrastructure for benchmarking java unit test generators," *CoRR*, vol. abs/2106.07520, 2021. arXiv: 2106.07520 (cit. on p. 262).

[67] T. Diamantopoulos, K. Thomopoulos, and A. Symeonidis, "Qualboa: Reusability-aware recommendations of source code components," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 488–491 (cit. on pp. 253, 257).

[68] J. Dietrich, H. Schole, L. Sui, and E. Tempero, "Xcorpus – an executable corpus of java programs," *Journal of Object Technology*, vol. 16, no. 4, 1:1–24, Aug. 2017 (cit. on pp. 5, 93, 96, 196, 249).

[69] E. W. Dijkstra *et al.*, *Notes on structured programming*, 1970 (cit. on p. 246).

[70] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005 (cit. on p. 93).

[71] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005 (cit. on pp. 177, 211, 261).

[72] F. Dong and S. G. Akl, "Scheduling algorithms for grid computing: State of the art and open problems," *School of Computing, Queen's University, Kingston, Ontario*, pp. 1–55, 2006 (cit. on p. 173).

[73] T. Dybå, V. B. Kampenes, and D. I. Sjøberg, "A systematic review of statistical power in software engineering experiments," *Information and Software Technology*, vol. 48, no. 8, pp. 745–755, 2006 (cit. on p. 211).

[74] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 422–431 (cit. on pp. 3, 273).

[75] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: Ultra-large-scale software repository and source-code mining," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, 2015 (cit. on pp. 3, 29, 131, 161, 253, 256, 262).

[76] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: Workshop on Dynamic Analysis*, Portland, OR, USA, May 2003, pp. 24–27 (cit. on pp. 4, 243, 245).

[77] ESEM. "Empirical Software Engineering and Measurement." (2022), [Online]. Available: `https://www.esem-conferences.org/` (visited on May 1, 2022) (cit. on p. 211).

[78] M. Fowler. "Continuous integration." (2006), [Online]. Available: `https://www.martinfowler.com/articles/continuousIntegration.html` (visited on May 1, 2022) (cit. on pp. 195, 203).

[79] M. Fowler, *Domain-specific languages*. Pearson Education, 2010 (cit. on p. 129).

[80] M. Fowler, *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004 (cit. on p. 42).

[81] G. Fraser, "A tutorial on using and extending the evosuite search-based test generator," in *Search-Based Software Engineering*, T. E. Colanzi and P. McMinn, Eds., Cham: Springer International Publishing, 2018, pp. 106–130 (cit. on pp. 215, 238).

[82] G. Fraser. "EvoSuite - Tutorial." (2022), [Online]. Available: `https://www.evosuite.org/documentation/tutorial-part-3/` (visited on May 1, 2022) (cit. on pp. 215, 224, 238).

[83] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, 2014 (cit. on pp. 177, 199, 202, 216, 222, 227, 237, 239, 250, 259).

[84] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, Szeged, Hungary: Association for Computing Machinery, 2011, 416–419 (cit. on pp. 25, 60, 180).

[85] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012 (cit. on p. 216).

[86] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? a controlled empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, pp. 1–49, 2015 (cit. on pp. 260, 261).

[87] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais, "The vocabulary problem in human-system communication," *Commun. ACM*, vol. 30, no. 11, 964–971, 1987 (cit. on pp. 108, 251).

[88] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, Leipzig, Germany: Association for Computing Machinery, 2008, pp. 321–330 (cit. on p. 256).

[89] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10, Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 147–156 (cit. on p. 256).

[90] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *ECOOP' 93 — Object-Oriented Programming*, O. M. Nierstrasz, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 406–431 (cit. on pp. 113, 120, 124, 142).

[91] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, *Elements of reusable object-oriented software*. Addison-Wesley Reading, Massachusetts, 1995, vol. 99 (cit. on p. 131).

[92] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *SIGPLAN Not.*, vol. 42, no. 10, 57–76, 2007 (cit. on p. 272).

[93] Git. "Git – fast version control." (2022), [Online]. Available: `https://git-scm.com/` (visited on May 1, 2022) (cit. on p. 93).

[94] GitHub. "The 2022 State of the Octoverse." (accessed 2022-12-01). (2022), [Online]. Available: `https://octoverse.github.com/` (cit. on p. 257).

[95] GitHub, Inc. "GitHub." (2022), [Online]. Available: `https://github.com/` (visited on May 1, 2022) (cit. on p. 94).

[96] N. Gold, J. Krinke, M. Harman, and D. Binkley, "Issues in clone classification for dataflow languages," in *Proceedings of the 4th International Workshop on Software Clones*, ser. IWSC '10, Cape Town, South Africa: Association for Computing Machinery, 2010, pp. 83–84 (cit. on p. 255).

[97] J. S. Gourlay, "A mathematical framework for the investigation of testing," *IEEE Trans. Softw. Eng.*, vol. 9, no. 6, 686–709, 1983 (cit. on p. 31).

[98] G. Gousios and D. Spinellis, "Ghtorrent: Github's data from a firehose," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, 2012, pp. 12–21 (cit. on p. 257).

[99] Gradle Inc. "Gradle Build Tool." (2022), [Online]. Available: `https://gradle.org/` (visited on May 1, 2022) (cit. on pp. 95, 130, 257).

[100] J. Gray, S. Chaudhuri, A. Bosworth, *et al.*, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data mining and knowledge discovery*, vol. 1, no. 1, pp. 29–53, 1997 (cit. on p. 87).

[101] L. D. Grazia and M. Pradel, "Code search: A survey of techniques for finding code," *ACM Comput. Surv.*, 2022 (cit. on p. 250).

[102] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 933–944 (cit. on p. 251).

[103] D. Güemes-Peña, C. López-Nozal, R. Marticorena-Sánchez, and J. Maudes-Raedo, "Emerging topics in mining software repositories," *Progress in Artificial Intelligence*, vol. 7, no. 3, pp. 237–247, 2018 (cit. on p. 243).

[104] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012, pp. 419–426 (cit. on p. 82).

[105] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977 (cit. on p. 63).

[106] D. Hamlet and J. Voas, "Faults on its sleeve: Amplifying software reliability testing," in *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '93, Cambridge, Massachusetts, USA: Association for Computing Machinery, 1993, 89–98 (cit. on p. 207).

[107] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001 (cit. on pp. 202, 271).

[108] J. L. Harrington, *Relational database design and implementation*. Morgan Kaufmann, 2016 (cit. on p. 83).

[109] A. E. Hassan, "The road ahead for mining software repositories," in *2008 Frontiers of Software Maintenance*, 2008, pp. 48–57 (cit. on p. 243).

[110] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995 (cit. on p. 63).

[111] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004 (cit. on p. 11).

[112] R. M. Hierons, K. Bogdanov, J. P. Bowen, *et al.*, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, Feb. 2009 (cit. on p. 41).

[113] A. Hindle, A. Wilson, K. Rasmussen, *et al.*, "Greenminer: A hardware based mining software repositories software energy consumption framework," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, Hyderabad, India: ACM, 2014, pp. 12–21 (cit. on pp. 257, 273).

[114] W. Hodges *et al.*, *A shorter model theory*. Cambridge university press, 1997 (cit. on p. 31).

[115] R. Holmes and R. J. Walker, "Systematizing pragmatic software reuse," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 4, 2013 (cit. on p. 117).

[116] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," in *Proceedings of the 14th International Conference on Software Engineering*, ser. ICSE '92, Melbourne, Australia: Association for Computing Machinery, 1992, 392–411 (cit. on p. 249).

[117] O. Hummel, "Semantic component retrieval in software engineering," Ph.D. dissertation, Universität Mannheim, 2008 (cit. on pp. 6, 106, 107, 109, 116, 121, 190, 251, 252).

[118] O. Hummel and C. Atkinson, "Using the web as a reuse repository," in *Reuse of Off-the-Shelf Components*, M. Morisio, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 298–311 (cit. on p. 250).

[119] O. Hummel and W. Janjic, "Test-driven reuse: Key to improving precision of search engines for software reuse," in *Finding Source Code on the Web for Remix and Reuse*, S. E. Sim and R. E. Gallardo-Valencia, Eds. Springer New York, 2013, pp. 227–250 (cit. on pp. 6, 113).

[120] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE Software*, vol. 25, no. 5, pp. 45–52, 2008 (cit. on pp. 106, 186, 252, 258).

[121] hello2morrow Inc. "Hello2morrow." (2022), [Online]. Available: `https://www.hello2morrow.com/products/sonargraph` (visited on May 1, 2022) (cit. on pp. 254, 273).

[122] JaCoCo Community. "JaCoCo." (2022), [Online]. Available: `https://www.jacoco.org/jacoco/trunk/doc/` (visited on May 1, 2022) (cit. on p. 143).

[123] W. Janjic, "Reuse-based test recommendation in software engineering," Ph.D. dissertation, 2014 (cit. on pp. 115, 188).

[124] JavaParser Community. "JavaParser - Analyse, transform and generate your Java codebase." (2022), [Online]. Available: `https://javaparser.org/` (visited on May 1, 2022) (cit. on p. 181).

[125] D. Jemerov and S. Isakova, *Kotlin in action*. Simon and Schuster, 2017 (cit. on p. 129).

[126] Jenkins Authors. "Jenkins." (2022), [Online]. Available: `https://www.jenkins.io/` (visited on May 1, 2022) (cit. on pp. 94, 130, 257).

[127] JetBrains. "IntelliJ IDEA." (2022), [Online]. Available: `https://www.jetbrains.com/idea/` (visited on May 1, 2022) (cit. on p. 186).

[128] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09, Chicago, IL, USA: Association for Computing Machinery, 2009, pp. 81–92 (cit. on pp. 255, 256).

[129] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Comput. Surv.*, vol. 36, no. 1, pp. 1–34, 2004 (cit. on pp. 132, 257).

[130] S. Josefsson, "The base16, base32, and base64 data encodings," RFC Editor, RFC 4648, 2006 (cit. on p. 21).

[131] E. Juergens, F. Deissenboeck, and B. Hummel, "Code similarities beyond copy paste," in *2010 14th European Conference on Software Maintenance and Reengineering*, 2010, pp. 78–87 (cit. on pp. 255, 256).

[132] JUnit. "JUnit." (2022), [Online]. Available: `https://junit.org/` (visited on May 1, 2022) (cit. on pp. 7, 15).

[133] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, San Jose, CA, USA: Association for Computing Machinery, 2014, 437–440 (cit. on pp. 177, 250).

[134] G. Karsai, H. Krahn, C. Pinkernell, *et al.*, *Design guidelines for domain specific languages*, 2014 (cit. on p. 147).

[135] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 295–306 (cit. on p. 197).

[136] M. Kessel and C. Atkinson, "A platform for diversity-driven test amplification," in *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2019, Tallinn, Estonia: Association for Computing Machinery, 2019, 35–41 (cit. on pp. 12, 207).

[137] M. Kessel and C. Atkinson, "Automatically curated data sets," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019, pp. 56–61 (cit. on p. 12).

[138] M. Kessel and C. Atkinson, "Diversity-driven unit test generation," *Journal of Systems and Software*, vol. 193, 2022 (cit. on pp. 12, 202, 203, 205, 227, 228, 233, 235).

[139] M. Kessel and C. Atkinson, "Integrating reuse into the rapid, continuous software engineering cycle through test-driven search," in *2018 IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering*, 2018, pp. 8–11 (cit. on pp. 12, 195, 203).

[140] M. Kessel and C. Atkinson, "Measuring the superfluous functionality in software components," in *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, ser. CBSE '15, Montréal, QC, Canada: Association for Computing Machinery, 2015, 11–20 (cit. on pp. 12, 62).

[141] M. Kessel and C. Atkinson, "On the efficacy of dynamic behavior comparison for judging functional equivalence," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019, pp. 193–203 (cit. on pp. 12, 107, 114, 118, 251, 269).

[142] M. Kessel and C. Atkinson, "Ranking software components for pragmatic reuse," in *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*, 2015, pp. 63–66 (cit. on pp. 12, 111, 253).

[143] M. Kessel and C. Atkinson, "Ranking software components for reuse based on non-functional properties," *Information Systems Frontiers*, vol. 18, no. 5, pp. 825–853, 2016 (cit. on pp. 12, 111, 180, 191, 253).

[144] K. Kim, D. Kim, T. F. Bissyandé, *et al.*, "Facoy: A code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, Gothenburg, Sweden: ACM, 2018, pp. 946–957 (cit. on pp. 117, 253, 256).

[145] B. Kitchenham, "What's up with software metrics? - a preliminary mapping study," *Journal of Systems and Software*, vol. 83, no. 1, pp. 37–51, 2010, SI: Top Scholars (cit. on p. 63).

[146] T. Kluyver, B. Ragan-Kelley, F. Pérez, *et al.*, "Jupyter notebooks ? a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Scmidt, Eds., IOS Press, 2016, pp. 87–90 (cit. on pp. 167, 171).

[147] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997 (cit. on pp. 15, 43).

[148] R. Koschke, "Survey of research on software clones," in *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007 (cit. on pp. 255, 256).

[149] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, 131–183, 1992 (cit. on pp. 3, 105, 185, 250).

[150] W. B. Langdon and M. Harman, "Optimizing existing software with genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, 2015 (cit. on p. 271).

[151] W. B. Langdon, S. Yoo, and M. Harman, "Inferring automatic test oracles," in *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*, 2017, pp. 5–6 (cit. on pp. 161, 270).

[152] C. Larman, *Applying UML and patterns: an introduction to object oriented analysis and design and interative development*. Pearson Education, 2012 (cit. on p. 51).

[153] T. Laurent, M. Papadakis, M. Kintis, *et al.*, "Assessing and improving the mutation testing practice of pit," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 430–435 (cit. on pp. 143, 261).

[154] O. A. Lazzarini Lemos, S. K. Bajracharya, and J. Ossher, "Codegenie: A tool for test-driven source code search," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, ser. OOPSLA '07, Montreal, Quebec, Canada: Association for Computing Machinery, 2007, 917–918 (cit. on pp. 6, 105, 186, 252, 258).

[155] O. A. Lazzarini Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes, "A test-driven approach to code search and its application to the reuse of auxiliary functionality," *Information and Software Technology*, vol. 53, no. 4, pp. 294–306, 2011, Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing (cit. on p. 252).

[156] H. Lechte, "Recommending reusable methods in IntelliJ by leveraging LASSO's code search capabilities," M.S. thesis, University of Mannheim, 2020 (cit. on p. 189).

[157] J. C. S. d. P. Leite, Y. Yu, L. Liu, E. S. K. Yu, and J. Mylopoulos, "Quality-based software reuse," in *Advanced Information Systems Engineering*, O. Pastor and J. Falcão e Cunha, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 535–550 (cit. on pp. 186, 193).

[158] O. A. L. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes, "Thesaurus-based automatic query expansion for interface-driven code search," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, Hyderabad, India: Association for Computing Machinery, 2014, 212–221 (cit. on p. 251).

[159] J. Lerner and J. Tirole, "The open source movement: Key research questions," *European Economic Review*, vol. 45, no. 4, pp. 819–826, 2001, 15th Annual Congress of the European Economic Association (cit. on p. 250).

[160] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 2, 2019 (cit. on pp. 64, 120, 244).

[161] Y. Lilis and A. Savidis, "A survey of metaprogramming languages," *ACM Comput. Surv.*, vol. 52, no. 6, 2019 (cit. on p. 120).

[162] E. Linstead, S. Bajracharya, T. Ngo, *et al.*, "Sourcerer: Mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, vol. 18, no. 2, pp. 300–336, 2009 (cit. on p. 177).

[163] J. Long, "Software reuse antipatterns," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 4, 68–76, 2001 (cit. on p. 193).

[164] C. V. Lopes, P. Maj, P. Martins, *et al.*, "Déjàvu: A map of code duplicates on github," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 2017 (cit. on pp. 3, 110).

[165] Y. Ma, T. Dey, C. Bogart, *et al.*, "World of code: Enabling a research workflow for mining and analyzing the universe of open source vcs data," *Empirical Software Engineering*, vol. 26, no. 2, p. 22, 2021 (cit. on p. 258).

[166] P. Maj, K. Siek, A. Kovalenko, and J. Vitek, "CodeDJ: Reproducible Queries over Large-Scale Software Repositories," in *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, A. Møller and M. Sridharan, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 194, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 6:1–6:24 (cit. on pp. 258, 273).

[167] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947 (cit. on p. 226).

[168] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. USA: Cambridge University Press, 2008 (cit. on pp. 21, 99, 101, 109, 158, 250, 251).

[169] S. T. March and V. C. Storey, "Design science in the information systems discipline: An introduction to the special issue on design science research," *MIS Quarterly*, vol. 32, no. 4, pp. 725–730, 2008 (cit. on p. 11).

[170] V. Markovtsev and W. Long, "Public git archive: A big code dataset for all," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, 34–37 (cit. on p. 257).

[171] P. Martins, R. Achar, and C. V. Lopes, "50k-c: A dataset of compilable, and compiled, java projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18, Gothenburg, Sweden: ACM, 2018, pp. 1–5 (cit. on p. 93).

[172] P. Martins, R. Achar, and C. V. Lopes, "50k-c: A dataset of compilable, and compiled, java projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, 1–5 (cit. on pp. 96, 177, 249).

[173] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976 (cit. on pp. 23, 63, 67).

[174] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 153–163 (cit. on pp. 4, 115, 202, 259).

[175] T. M. Meyers and D. Binkley, "An empirical study of slice-based cohesion and coupling metrics," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 1, 2007 (cit. on p. 248).

[176] H. Mili, F. Mili, and A. Mili, "Reusing software: Issues and research directions," *IEEE Transactions on Software Engineering*, vol. 21, no. 6, pp. 528–562, 1995 (cit. on pp. 186, 250).

[177] G. A. Miller, "Wordnet: A lexical database for english," *Commun. ACM*, vol. 38, no. 11, 39–41, 1995 (cit. on pp. 109, 110).

[178] R. Mugridge and W. Cunningham, *Fit for developing software: framework for integrated tests*. Pearson Education, 2005 (cit. on p. 246).

[179] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, Saint Petersburg, Russia: Association for Computing Machinery, 2013, 466–476 (cit. on pp. 9, 177, 220).

[180] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, 31–88, 2001 (cit. on p. 153).

[181] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr., *et al.*, "A graph-based approach to api usage adaptation," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10, Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 302–321 (cit. on p. 252).

[182] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query expansion based on crowd knowledge for code search," *IEEE Transactions on Services Computing*, vol. 9, no. 5, pp. 771–783, 2016 (cit. on p. 251).

[183] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer Science & Business Media, 2015 (cit. on p. 243).

[184] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07, USA: IEEE Computer Society, 2007, pp. 75–84 (cit. on pp. 25, 60, 115, 180, 259).

[185] J. Palsberg and C. V. Lopes, "Njr: A normalized java resource," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ser. ISSTA '18, Amsterdam, Netherlands: Association for Computing Machinery, 2018, 100–106 (cit. on pp. 93, 96, 98, 177, 196, 249).

[186] Pandas Authors. "Pandas." (2022), [Online]. Available: `https://pandas.pydata.org/` (visited on May 1, 2022) (cit. on p. 78).

[187] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017 (cit. on pp. 202, 216, 222, 231, 235, 259).

[188] M. Papadakis, M. Kintis, J. Zhang, *et al.*, "Chapter six - mutation testing advances: An analysis and survey," in ser. Advances in Computers, A. M. Memon, Ed., vol. 112, Elsevier, 2019, pp. 275–378 (cit. on p. 260).

[189] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995 (cit. on pp. 50, 54).

[190] A. Podgurski and L. Pierce, "Behavior sampling: A technique for automated retrieval of reusable components," in *Proceedings of the 14th International Conference on Software Engineering*, ser. ICSE '92, Melbourne, Australia: ACM, 1992, pp. 349–361 (cit. on p. 114).

[191] A. Podgurski and L. Pierce, "Retrieving reusable software by sampling behavior," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 3, pp. 286–303, Jul. 1993 (cit. on pp. 6, 113, 114, 251, 269).

[192] Python Software Foundation. “Python Package Index.” (2022), [Online]. Available: https://pypi.org/ (visited on May 1, 2022) (cit. on p. 94).

[193] Python Software Foundation. “Unit Testing Framework.” (2022), [Online]. Available: https://docs.python.org/3/library/unittest.html (visited on May 1, 2022) (cit. on p. 7).

[194] S. Raemaekers, A. van Deursen, and J. Visser, “Semantic versioning and impact of breaking changes in the maven repository,” *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017 (cit. on pp. 98, 176).

[195] S. Raemaekers, A. van Deursen, and J. Visser, “The maven repository dataset of metrics, changes, and dependencies,” in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 221–224 (cit. on pp. 98, 175).

[196] M. A. Rahman and Y. Wang, “Optimizing intersection-over-union in deep neural networks for image segmentation,” in *Advances in Visual Computing*, G. Bebis, R. Boyle, B. Parvin, *et al.*, Eds., Cham: Springer International Publishing, 2016, pp. 234–244 (cit. on p. 154).

[197] R. Ramakrishnan, J. Gehrke, and J. Gehrke, *Database management systems*. McGraw-Hill New York, 2003, vol. 3 (cit. on pp. 83, 86).

[198] D. Rattan, R. Bhatia, and M. Singh, “Software clone detection: A systematic review,” *Information and Software Technology*, vol. 55, no. 7, pp. 1165 –1199, 2013 (cit. on p. 255).

[199] S. P. Reiss, “Towards creating test cases using code search,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 436–440 (cit. on p. 115).

[200] S. P. Reiss, “Semantics-based code search,” in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 243–253 (cit. on pp. 6, 252, 253).

[201] H. G. Rice, “Classes of recursively enumerable sets and their decision problems,” *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953 (cit. on pp. 4, 23, 40, 244).

[202] S. Robertson and H. Zaragoza, “The probabilistic relevance framework: Bm25 and beyond,” *Found. Trends Inf. Retr.*, vol. 3, no. 4, Apr. 2009 (cit. on p. 110).

[203] M. Robillard, R. Walker, and T. Zimmermann, “Recommendation systems for software engineering,” *IEEE Software*, vol. 27, no. 4, pp. 80–86, 2010 (cit. on p. 3).

[204] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, *Recommendation Systems in Software Engineering*. Springer, 2014 (cit. on pp. 105, 185, 251).

[205] S. Romano, C. Vendome, G. Scanniello, and D. Poshyvanyk, “A multi-study investigation into dead code,” *IEEE Transactions on Software Engineering*, vol. 46, no. 1, pp. 71–99, 2020 (cit. on p. 62).

[206] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009 (cit. on pp. 101, 255).

[207] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007 (cit. on pp. 146, 201, 255).

[208] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 354–365 (cit. on p. 256).

[209] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1157–1168 (cit. on pp. 3, 257).

[210] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, "Data-driven equivalence checking," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA'13, Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 391–406 (cit. on p. 245).

[211] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 9–19 (cit. on pp. 212, 226).

[212] S. E. Sim and R. E. Gallardo-Valencia, *Finding Source Code on the Web for Remix and Reuse*. Springer Publishing Company, Incorporated, 2015 (cit. on pp. 3, 6, 105, 110, 176, 185, 251).

[213] F. Simon, C. Lewerentz, and W. Bischofberger, "Software quality assessments for system, architecture, design and code," in *Software Quality and Software Testing in Internet Times*, D. Meyerhoff, B. Laibarra, R. van der Pouw Kraan, and A. Wallet, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 230–249 (cit. on p. 254).

[214] D. Sjoeberg, J. Hannay, O. Hansen, *et al.*, "A survey of controlled experiments in software engineering," *IEEE Transactions on Software Engineering*, vol. 31, no. 9, pp. 733–753, 2005 (cit. on p. 211).

[215] Slashdot Media. "SourceForge." (2022), [Online]. Available: `https://sourceforge.net/` (visited on May 1, 2022) (cit. on p. 94).

[216] SonarSource. "SonarQube." (accessed 2022-05-01). (2022), [Online]. Available: `https://www.sonarqube.org/` (cit. on pp. 254, 273).

[217] Sonatype. "Maven Central." (2022), [Online]. Available: `http://search.maven.org` (visited on May 1, 2022) (cit. on pp. 5, 94, 95, 167, 175).

[218] M. Staats, M. W. Whalen, and M. P. Heimdahl, "Programs, tests, and oracles: The foundations of testing revisited," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, New York, NY, USA: Association for Computing Machinery, 2011, pp. 391–400 (cit. on p. 31).

[219] Stack Exchange, Inc. "Stack Overflow." (2022), [Online]. Available: `https://stackoverflow.com/` (visited on May 1, 2022) (cit. on p. 94).

[220] G. Steele, *Common LISP: the language*. Elsevier, 1990 (cit. on p. 84).

[221] K. T. Stolee, S. Elbaum, and D. Dobos, "Solving the search for source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 3, 26:1–26:45, Jun. 2014 (cit. on p. 252).

[222] M.-A. Storey, C. Treude, A. van Deursen, and L.-T. Cheng, "The impact of social media on software engineering practices and tools," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10, Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, 359–364 (cit. on p. 94).

[223] M. Suzuki, A. Carvalho de Paula, E. Guerra, C. V. Lopes, and O. A. Lazzarini Lemos, "An exploratory study of functional redundancy in code repositories," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2017, pp. 31–40 (cit. on p. 269).

[224] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 476–480 (cit. on p. 256).

[225] J. Svajlenko and C. K. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 596–600 (cit. on pp. 255, 256, 263).

[226] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 131–140 (cit. on pp. 96, 177, 250, 255, 263).

[227] A. Tahir and S. G. MacDonell, "A systematic mapping study on dynamic metrics and software quality," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 326–335 (cit. on p. 63).

[228] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 1, 35–42, 2005 (cit. on p. 206).

[229] E. Tempero, C. Anslow, J. Dietrich, *et al.*, "The qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference*, 2010, pp. 336–345 (cit. on pp. 177, 249).

[230] R. Terra, L. F. Miranda, M. T. Valente, and R. S. Bigonha, "Qualitas.class corpus: A compiled version of the qualitas corpus," *SIGSOFT Softw. Eng. Notes*, vol. 38, no. 5, pp. 1–4, Aug. 2013 (cit. on p. 93).

[231] R. Terra, L. F. Miranda, M. T. Valente, and R. S. Bigonha, "Qualitas.class corpus: A compiled version of the qualitas corpus," *SIGSOFT Softw. Eng. Notes*, vol. 38, no. 5, 1–4, 2013 (cit. on p. 177).

[232] The Apache Software Foundation. "Apache Ignite." (2022), [Online]. Available: `https://ignite.apache.org/` (visited on May 1, 2022) (cit. on p. 169).

[233] The Apache Software Foundation. "Apache Maven Project." (2022), [Online]. Available: `https://maven.apache.org` (visited on May 1, 2022) (cit. on pp. 5, 95).

[234] The Apache Software Foundation. “Apache Solr Project.” (2022), [Online]. Available: https://solr.apache.org/ (visited on May 1, 2022) (cit. on pp. 23, 101).

[235] The Apache Software Foundation. “Apache Spark.” (2022), [Online]. Available: https://spark.apache.org/ (visited on May 1, 2022) (cit. on pp. 151, 248).

[236] The Apache Software Foundation. “Apache Zeppelin.” (2022), [Online]. Available: https://zeppelin.apache.org/ (visited on May 1, 2022) (cit. on p. 167).

[237] The Apache Software Foundation. “Groovy - Domain-Specific Languages.” (2022), [Online]. Available: http://docs.groovy-lang.org/docs/latest/html/documentation/core-domain-specific-languages.html (visited on May 1, 2022) (cit. on p. 143).

[238] The Apache Software Foundation. “Subversion.” (2022), [Online]. Available: https://subversion.apache.org/ (visited on May 1, 2022) (cit. on p. 93).

[239] The R Foundation. “The R Project for Statistical Computing.” (2022), [Online]. Available: https://www.r-project.org/ (visited on May 1, 2022) (cit. on pp. 20, 151).

[240] Tidyverse Community. “Tidyverse.” (2022), [Online]. Available: https://www.tidyverse.org/ (visited on May 1, 2022) (cit. on p. 248).

[241] L. D. Toffola, C.-A. Staicu, and M. Pradel, “Saying ‘hi!’ is not enough: Mining inputs for effective test generation,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 44–49 (cit. on p. 115).

[242] A. M. Turing, “On computable numbers, with an application to the entscheidungsproblem. a correction,” *Proceedings of the London Mathematical Society*, vol. 2, no. 1, pp. 544–546, 1938 (cit. on pp. 4, 244).

[243] M. Van Steen and A Tanenbaum, *Distributed systems principles and paradigms (2nd Edition)*. Prentice-Hall, Inc., 2007 (cit. on p. 82).

[244] A. Vargha and H. D. Delaney, “A critique and improvement of the cl common language effect size statistics of mcgraw and wong,” *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000 (cit. on p. 226).

[245] VMware, Inc. “Spring.” (2022), [Online]. Available: https://spring.io/ (visited on May 1, 2022) (cit. on pp. 131, 167).

[246] S. Vogl, S. Schweikl, G. Fraser, *et al.*, “Evosuite at the sbst 2021 tool competition,” in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, IEEE, 2021, pp. 28–29 (cit. on p. 259).

[247] A. Von Mayrhauser and A. Vans, “Program comprehension during software maintenance and evolution,” *Computer*, vol. 28, no. 8, pp. 44–55, 1995 (cit. on p. 61).

[248] W3C. “XML Path Language (XPath) 3.1.” (2017), [Online]. Available: https://www.w3.org/TR/xpath-31/ (visited on May 1, 2022) (cit. on p. 80).

[249] Y. Wang, Y. Feng, R. Martins, *et al.*, "Hunter: Next-generation code reuse for Java," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, Seattle, WA, USA: ACM, 2016, pp. 1028–1032 (cit. on pp. 121, 252).

[250] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984 (cit. on pp. 7, 67, 248).

[251] C. Wohlin, P. Runeson, M. Höst, *et al.*, *Experimentation in software engineering*. Springer Science & Business Media, 2012 (cit. on pp. 9, 93, 211, 213, 214).

[252] D. Wolpert and W. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997 (cit. on pp. 202, 216).

[253] M. Wu, P. Wang, K. Yin, *et al.*, "Lvmapper: A large-variance clone detector using sequencing alignment approach," *IEEE Access*, vol. 8, pp. 27 986–27 997, 2020 (cit. on p. 255).

[254] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, 2005 (cit. on pp. 7, 67, 249).

[255] A. Zaidman, "Scalability solutions for program comprehension through dynamic analysis," in *Conference on Software Maintenance and Reengineering (CSMR'06)*, 2006, 4 pp.–330 (cit. on p. 71).

[256] A. M. Zaremski and J. M. Wing, "Signature matching: A tool for using software libraries," *ACM Trans. Softw. Eng. Methodol.*, vol. 4, no. 2, 1995 (cit. on p. 251).

[257] P. Zhang and S. Elbaum, "Amplifying tests to validate exception handling code," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 595–605 (cit. on p. 207).